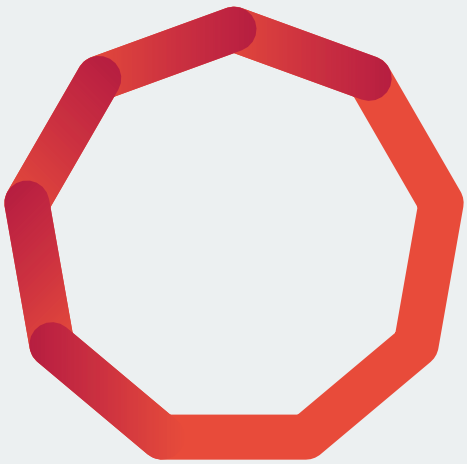


Distributed Systems

the source of truth

essential edition

Feel free to contribute at `https://github.com/alexprut/distributed-systems`

First release, August 2018, version 1.0

Contents

Introduction **5**

1 Time, Clocks, and the Ordering of Events in a Distributed System **7**

2 Distributed Snapshots: Determining Global States of Distributed Systems **17**

3 The Byzantine Generals Problem **31**

4 The Part-Time Parliament **53**

5 Paxos Made Simple **87**

6 Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web **103**

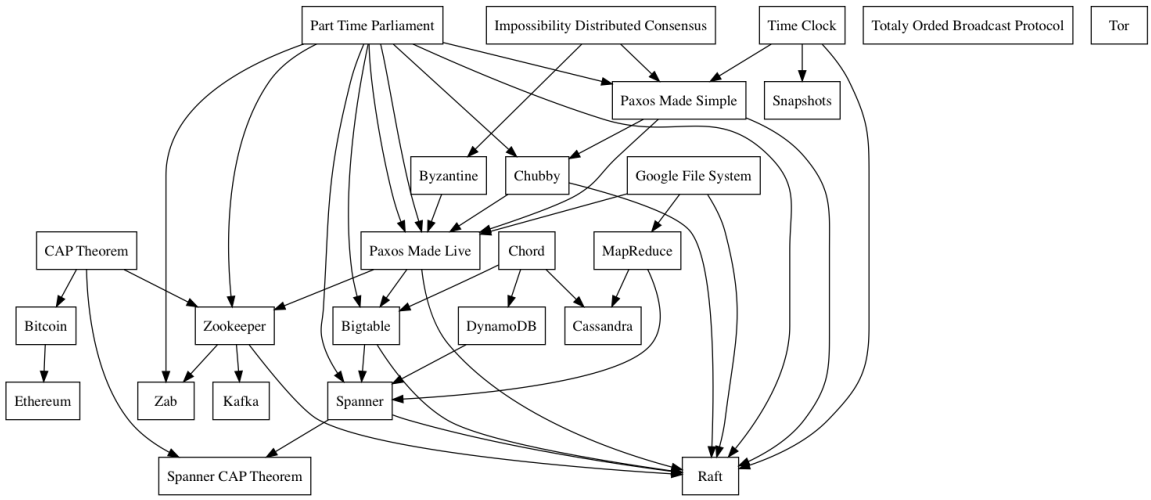
7 MapReduce: Simplified Data Processing on Large Clusters **117**

8 Paxos Made Live - An Engineering Perspective **131**

9 Bitcoin: A Peer-to-Peer Electronic Cash System **149**

Introduction

Both in the libraries and on the internet plenty of resources are available regarding the **Distributed Systems** topic. All of them are just a better explanation of the same exact source of truth, i.e. academic papers published by the brightest minds in the world. This book is just an ordered collection of all those publications. Below is depicted the order in which you can read the papers. The primary order is based on the citations dependency in the publications, secondary order is based on the year of publication and finally the last criteria is based on the topic dependency.



Chapter 1

Time, Clocks, and the Ordering of Events in a Distributed System

Authors:
Leslie Lamport

Publication year: 1978

Operating Systems

R. Stockton Gaines
Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

© 1978 ACM 0001-0782/78/0700-0558 $00.75

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event a happened before an event b if a happened at an earlier time than b. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Fig. 1.

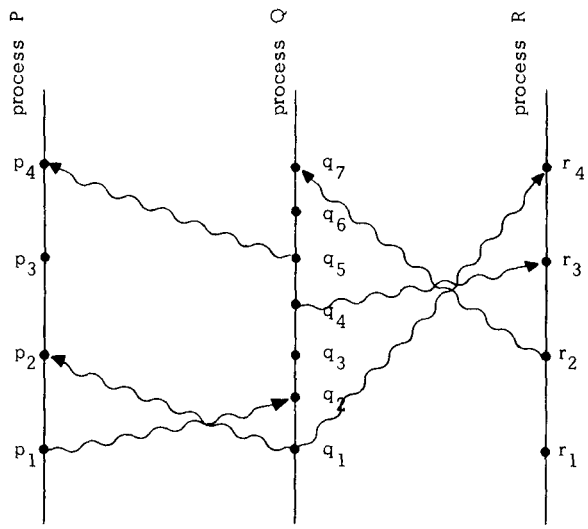
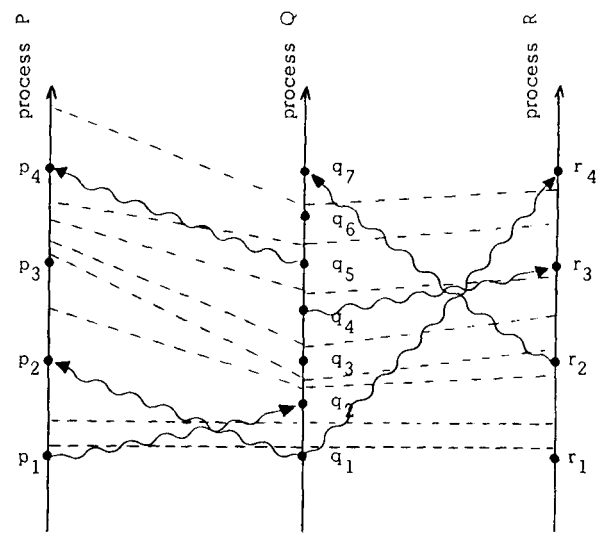


Fig. 2.



event. We are assuming that the events of a process form a sequence, where a occurs before b in this sequence if a happens before b. In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process.[1] It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

We assume that sending or receiving a message is an event in a process. We can then define the "happened before" relation, denoted by "$\rightarrow$", as follows.

Definition. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

We assume that $a \not\rightarrow a$ for any event a. (Systems in which an event can happen before itself do not seem to be physically meaningful.) This implies that $\rightarrow$ is an irreflexive partial ordering on the set of all events in the system.

It is helpful to view this definition in terms of a "space-time diagram" such as Figure 1. The horizontal direction represents space, and the vertical direction represents time—later times being higher than earlier ones. The dots denote events, the vertical lines denote processes, and the wavy lines denote messages.[2] It is easy to see that $a \rightarrow b$ means that one can go from a to b in the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_4$ in Figure 1.

Another way of viewing the definition is to say that $a \rightarrow b$ means that it is possible for event a to causally affect event b. Two events are concurrent if neither can causally affect the other. For example, events p_3 and q_3 of Figure 1 are concurrent. Even though we have drawn the diagram to imply that q_3 occurs at an earlier physical time than p_3, process P cannot know what process Q did at q_3 until it receives the message at p_4. (Before event p_4, P could at most know what Q was *planning* to do at q_3.)

This definition will appear quite natural to the reader familiar with the invariant space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is defined in terms of messages that *could* be sent. However, we have taken the more pragmatic approach of only considering messages that actually *are* sent. We should be able to determine if a system performed correctly by knowing only those events which *did* occur, without knowing which events *could* have occurred.

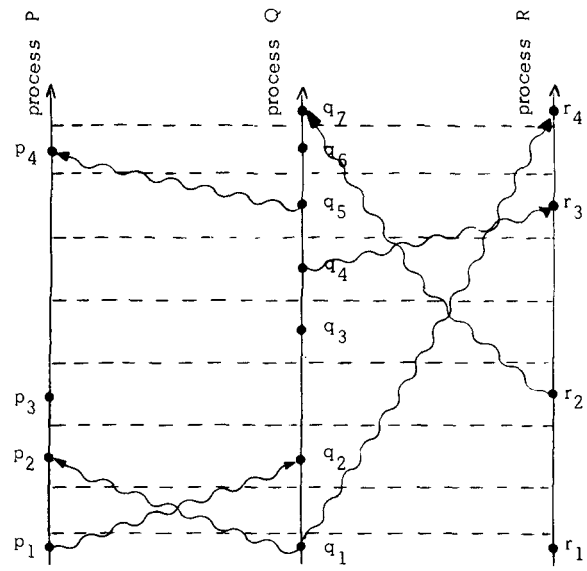
Logical Clocks

We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. More precisely, we define a clock C_i for each process P_i to be a function which assigns a number $C_i\langle a\rangle$ to any event a in that process. The entire system of clocks is represented by the function C which assigns to any event b the number $C\langle b\rangle$, where $C\langle b\rangle = C_j\langle b\rangle$ if b is an event in process P_j. For now, we make no assumption about the relation of the numbers $C_i\langle a\rangle$ to physical time, so we can think of the clocks C_i as logical rather than physical clocks. They may be implemented by counters with no actual timing mechanism.

[1] The choice of what constitutes an event affects the ordering of events in a process. For example, the receipt of a message might denote the setting of an interrupt bit in a computer, or the execution of a subprogram to handle that interrupt. Since interrupts need not be handled in the order that they occur, this choice will affect the ordering of a process' message-receiving events.

[2] Observe that messages may be received out of order. We allow the sending of several messages to be a single event, but for convenience we will assume that the receipt of a single message does not coincide with the sending or receipt of any other message.

Fig. 3.



We now consider what it means for such a system of clocks to be correct. We cannot base our definition of correctness on physical time, since that would require introducing clocks which keep physical time. Our definition must be based on the order in which events occur. The strongest reasonable condition is that if an event a occurs before another event b, then a should happen at an earlier time than b. We state this condition more formally as follows.

Clock Condition. For any events a, b:

$$\text{if } a \rightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time. In Figure 1, p_2 and p_3 are both concurrent with q_3, so this would mean that they both must occur at the same time as q_3, which would contradict the Clock Condition because $p_2 \rightarrow p_3$.

It is easy to see from our definition of the relation "$\rightarrow$" that the Clock Condition is satisfied if the following two conditions hold.

C1. If a and b are events in process P_i, and a comes before b, then $C_i\langle a\rangle < C_i\langle b\rangle$.

C2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.

Let us consider the clocks in terms of a space-time diagram. We imagine that a process' clock "ticks" through every number, with the ticks occurring between the process' events. For example, if a and b are consecutive events in process P_i with $C_i\langle a\rangle = 4$ and $C_i\langle b\rangle = 7$, then clock ticks 5, 6, and 7 occur between the two events. We draw a dashed "tick line" through all the like-numbered ticks of the different processes. The space-time diagram of Figure 1 might then yield the picture in Figure 2. Condition C1 means that there must be a tick line between any two events on a process line, and condition C2 means that every message line must cross a tick line. From the pictorial meaning of $\rightarrow$, it is easy to see why these two conditions imply the Clock Condition.

We can consider the tick lines to be the time coordinate lines of some Cartesian coordinate system on space-time. We can redraw Figure 2 to straighten these coordinate lines, thus obtaining Figure 3. Figure 3 is a valid alternate way of representing the same system of events as Figure 2. Without introducing the concept of physical time into the system (which requires introducing physical clocks), there is no way to decide which of these pictures is a better representation.

The reader may find it helpful to visualize a two-dimensional spatial network of processes, which yields a three-dimensional space-time diagram. Processes and messages are still represented by lines, but tick lines become two-dimensional surfaces.

Let us now assume that the processes are algorithms, and the events represent certain actions during their execution. We will show how to introduce clocks into the processes which satisfy the Clock Condition. Process P_i's clock is represented by a register C_i, so that $C_i\langle a\rangle$ is the value contained by C_i during the event a. The value of C_i will change between events, so changing C_i does not itself constitute an event.

To guarantee that the system of clocks satisfies the Clock Condition, we will insure that it satisfies conditions C1 and C2. Condition C1 is simple; the processes need only obey the following implementation rule:

IR1. Each process P_i increments C_i between any two successive events.

To meet condition C2, we require that each message m contain a *timestamp* T_m which equals the time at which the message was sent. Upon receiving a message time-stamped T_m, a process must advance its clock to be later than T_m. More precisely, we have the following rule.

IR2. (a) If event a is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i\langle a\rangle$. (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

In IR2(b) we consider the event which represents the receipt of the message m to occur after the setting of C_j. (This is just a notational nuisance, and is irrelevant in any actual implementation.) Obviously, IR2 insures that C2 is satisfied. Hence, the simple implementation rules IR1 and IR2 imply that the Clock Condition is satisfied, so they guarantee a correct system of logical clocks.

Ordering the Events Totally

We can use a system of clocks satisfying the Clock Condition to place a total ordering on the set of all system events. We simply order the events by the times

at which they occur. To break ties, we use any arbitrary total ordering $<$ of the processes. More precisely, we define a relation $\Rightarrow$ as follows: if a is an event in process P_i and b is an event in process P_j, then $a \Rightarrow b$ if and only if either (i) $C_i\langle a\rangle < C_j\langle b\rangle$ or (ii) $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i < P_j$. It is easy to see that this defines a total ordering, and that the Clock Condition implies that if $a \rightarrow b$ then $a \Rightarrow b$. In other words, the relation $\Rightarrow$ is a way of completing the "happened before" partial ordering to a total ordering.[3]

The ordering $\Rightarrow$ depends upon the system of clocks C_i, and is not unique. Different choices of clocks which satisfy the Clock Condition yield different relations $\Rightarrow$. Given any total ordering relation $\Rightarrow$ which extends $\rightarrow$, there is a system of clocks satisfying the Clock Condition which yields that relation. It is only the partial ordering $\rightarrow$ which is uniquely determined by the system of events.

Being able to totally order the events can be very useful in implementing a distributed system. In fact, the reason for implementing a correct system of logical clocks is to obtain such a total ordering. We will illustrate the use of this total ordering of events by solving the following version of the mutual exclusion problem. Consider a system composed of a fixed collection of processes which share a single resource. Only one process can use the resource at a time, so the processes must synchronize themselves to avoid conflict. We wish to find an algorithm for granting the resource to a process which satisfies the following three conditions: (I) A process which has been granted the resource must release it before it can be granted to another process. (II) Different requests for the resource must be granted in the order in which they are made. (III) If every process which is granted the resource eventually releases it, then every request is eventually granted.

We assume that the resource is initially granted to exactly one process.

These are perfectly natural requirements. They precisely specify what it means for a solution to be correct.[4] Observe how the conditions involve the ordering of events. Condition II says nothing about which of two concurrently issued requests should be granted first.

It is important to realize that this is a nontrivial problem. Using a central scheduling process which grants requests in the order they are received will not work, unless additional assumptions are made. To see this, let P_0 be the scheduling process. Suppose P_1 sends a request to P_0 and then sends a message to P_2. Upon receiving the latter message, P_2 sends a request to P_0. It is possible for P_2's request to reach P_0 before P_1's request does. Condition II is then violated if P_2's request is granted first.

To solve the problem, we implement a system of clocks with rules IR1 and IR2, and use them to define a total ordering $\Rightarrow$ of all events. This provides a total ordering of all request and release operations. With this ordering, finding a solution becomes a straightforward exercise. It just involves making sure that each process learns about all other processes' operations.

To simplify the problem, we make some assumptions. They are not essential, but they are introduced to avoid distracting implementation details. We assume first of all that for any two processes P_i and P_j, the messages sent from P_i to P_j are received in the same order as they are sent. Moreover, we assume that every message is eventually received. (These assumptions can be avoided by introducing message numbers and message acknowledgment protocols.) We also assume that a process can send messages directly to every other process.

Each process maintains its own *request queue* which is never seen by any other process. We assume that the request queues initially contain the single message T_0:P_0 *requests resource*, where P_0 is the process initially granted the resource and T_0 is less than the initial value of any clock.

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.

2. When process P_j receives the message T_m:P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.[5]

3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.

4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.

5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m:P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $\Rightarrow$. (To define the relation "$\Rightarrow$" for messages, we identify a message with the event of sending it.) (ii) P_i has received a message from every other process timestamped later than T_m.[6]

Note that conditions (i) and (ii) of rule 5 are tested locally by P_i.

It is easy to verify that the algorithm defined by these rules satisfies conditions I–III. First of all, observe that condition (ii) of rule 5, together with the assumption that messages are received in order, guarantees that P_i has learned about all requests which preceded its current

[3] The ordering $<$ establishes a priority among the processes. If a "fairer" method is desired, then $<$ can be made a function of the clock value. For example, if $C_i\langle a\rangle = C_j\langle b\rangle$ and $j < i$, then we can let $a \Rightarrow b$ if $j < C_i\langle a\rangle \bmod N \leq i$, and $b \Rightarrow a$ otherwise; where N is the total number of processes.

[4] The term "eventually" should be made precise, but that would require too long a diversion from our main topic.

[5] This acknowledgment message need not be sent if P_j has already sent a message to P_i timestamped later than T_m.

[6] If $P_i < P_j$, then P_i need only have received a message timestamped $\geq T_m$ from P_j.

request. Since rules 3 and 4 are the only ones which delete messages from the request queue, it is then easy to see that condition I holds. Condition II follows from the fact that the total ordering $\Rightarrow$ extends the partial ordering $\rightarrow$. Rule 2 guarantees that after P_i requests the resource, condition (ii) of rule 5 will eventually hold. Rules 3 and 4 imply that if each process which is granted the resource eventually releases it, then condition (i) of rule 5 will eventually hold, thus proving condition III.

This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*, consisting of a set $\boldsymbol{C}$ of possible commands, a set $\boldsymbol{S}$ of possible states, and a function $\boldsymbol{e}: \boldsymbol{C} \times \boldsymbol{S} \rightarrow \boldsymbol{S}$. The relation $\boldsymbol{e}(C, S) = S'$ means that executing the command C with the machine in state S causes the machine state to change to S'. In our example, the set $\boldsymbol{C}$ consists of all the commands P_i *requests resource* and P_i *releases resource*, and the state consists of a queue of waiting *request* commands, where the request at the head of the queue is the currently granted one. Executing a *request* command adds the request to the tail of the queue, and executing a *release* command removes a command from the queue.[7]

Each process independently simulates the execution of the State Machine, using the commands issued by all the processes. Synchronization is achieved because all processes order the commands according to their timestamps (using the relation $\Rightarrow$), so each process uses the same sequence of commands. A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T. The precise algorithm is straightforward, and we will not bother to describe it.

This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, the resulting algorithm requires the active participation of all the processes. A process must know all the commands issued by other processes, so that the failure of a single process will make it impossible for any other process to execute State Machine commands, thereby halting the system.

The problem of failure is a difficult one, and it is beyond the scope of this paper to discuss it in any detail. We will just observe that the entire concept of failure is only meaningful in the context of physical time. Without physical time, there is no way to distinguish a failed process from one which is just pausing between events. A user can tell that a system has "crashed" only because he has been waiting too long for a response. A method which works despite the failure of individual processes or communication lines is described in [3].

[7] If each process does not strictly alternate *request* and *release* commands, then executing a *release* command could delete zero, one, or more than one request from the queue.

Anomalous Behavior

Our resource scheduling algorithm ordered the requests according to the total ordering $\Rightarrow$. This permits the following type of "anomalous behavior." Consider a nationwide system of interconnected computers. Suppose a person issues a request A on a computer A, and then telephones a friend in another city to have him issue a request B on a different computer B. It is quite possible for request B to receive a lower timestamp and be ordered before request A. This can happen because the system has no way of knowing that A actually preceded B, since that precedence information is based on messages external to the system.

Let us examine the source of the problem more closely. Let $\mathcal{S}$ be the set of all system events. Let us introduce a set of events which contains the events in $\mathcal{S}$ together with all other relevant external events, such as the phone calls in our example. Let $\rightarrow$ denote the "happened before" relation for $\underline{\mathcal{S}}$. In our example, we had A $\rightarrow$ B, but A $\not\rightarrow$ B. It is obvious that no algorithm based entirely upon events in $\mathcal{S}$, and which does not relate those events in any way with the other events in $\underline{\mathcal{S}}$, can guarantee that request A is ordered before request B.

There are two possible ways to avoid such anomalous behavior. The first way is to explicitly introduce into the system the necessary information about the ordering $\rightarrow$. In our example, the person issuing request A could receive the timestamp T_A of that request from the system. When issuing request B, his friend could specify that B be given a timestamp later than T_A. This gives the user the responsibility for avoiding anomalous behavior.

The second approach is to construct a system of clocks which satisfies the following condition.

Strong Clock Condition. For any events a, b in $\mathcal{S}$:

$$\text{if } a \rightarrow b \text{ then } C\langle a\rangle < C\langle b\rangle.$$

This is stronger than the ordinary Clock Condition because $\rightarrow$ is a stronger relation than $\rightarrow$. It is not in general satisfied by our logical clocks.

Let us identify $\underline{\mathcal{S}}$ with some set of "real" events in physical space-time, and let $\rightarrow$ be the partial ordering of events defined by special relativity. One of the mysteries of the universe is that it is possible to construct a system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition. We can therefore use physical clocks to eliminate anomalous behavior. We now turn our attention to such clocks.

Physical Clocks

Let us introduce a physical time coordinate into our space-time picture, and let $C_i(t)$ denote the reading of the clock C_i at physical time t.[8] For mathematical con-

[8] We will assume a Newtonian space-time. If the relative motion of the clocks or gravitational effects are not negligible, then $C_i(t)$ must be deduced from the actual clock reading by transforming from proper time to the arbitrarily chosen time coordinate.

venience, we assume that the clocks run continuously rather than in discrete "ticks." (A discrete clock can be thought of as a continuous one in which there is an error of up to $\frac{1}{2}$ "tick" in reading it.) More precisely, we assume that $C_i(t)$ is a continuous, differentiable function of t except for isolated jump discontinuities where the clock is reset. Then $dC_i(t)/dt$ represents the rate at which the clock is running at time t.

In order for the clock C_i to be a true physical clock, it must run at approximately the correct rate. That is, we must have $dC_i(t)/dt \approx 1$ for all t. More precisely, we will assume that the following condition is satisfied:

PC1. There exists a constant $\kappa \ll 1$ such that for all i: $|dC_i(t)/dt - 1| < \kappa$.

For typical crystal controlled clocks, $\kappa \leq 10^{-6}$.

It is not enough for the clocks individually to run at approximately the correct rate. They must be synchronized so that $C_i(t) \approx C_j(t)$ for all i, j, and t. More precisely, there must be a sufficiently small constant ϵ so that the following condition holds:

PC2. For all i, j: $|C_i(t) - C_j(t)| < \epsilon$.

If we consider vertical distance in Figure 2 to represent physical time, then PC2 states that the variation in height of a single tick line is less than ϵ.

Since two different clocks will never run at exactly the same rate, they will tend to drift further and further apart. We must therefore devise an algorithm to insure that PC2 always holds. First, however, let us examine how small κ and ϵ must be to prevent anomalous behavior. We must insure that the system $\underline{\mathcal{S}}$ of relevant physical events satisfies the Strong Clock Condition. We assume that our clocks satisfy the ordinary Clock Condition, so we need only require that the Strong Clock Condition holds when a and b are events in $\underline{\mathcal{S}}$ with $a \not\rightarrow b$. Hence, we need only consider events occurring in different processes.

Let μ be a number such that if event a occurs at physical time t and event b in another process satisfies $a \rightarrow b$, then b occurs later than physical time $t + \mu$. In other words, μ is less than the shortest transmission time for interprocess messages. We can always choose μ equal to the shortest distance between processes divided by the speed of light. However, depending upon how messages in $\underline{\mathcal{S}}$ are transmitted, μ could be significantly larger.

To avoid anomalous behavior, we must make sure that for any i, j, and t: $C_i(t + \mu) - C_j(t) > 0$. Combining this with PC1 and 2 allows us to relate the required smallness of κ and ϵ to the value of μ as follows. We assume that when a clock is reset, it is always set forward and never back. (Setting it back could cause C1 to be violated.) PC1 then implies that $C_i(t + \mu) - C_i(t) > (1 - \kappa)\mu$. Using PC2, it is then easy to deduce that $C_i(t + \mu) - C_j(t) > 0$ if the following inequality holds:

$$\epsilon/(1 - \kappa) \leq \mu.$$

This inequality together with PC1 and PC2 implies that anomalous behavior is impossible.

We now describe our algorithm for insuring that PC2 holds. Let m be a message which is sent at physical time t and received at time t'. We define $\nu_m = t' - t$ to be the *total delay* of the message m. This delay will, of course, not be known to the process which receives m. However, we assume that the receiving process knows some *minimum delay* $\mu_m \geq 0$ such that $\mu_m \leq \nu_m$. We call $\xi_m = \nu_m - \mu_m$ the *unpredictable delay* of the message.

We now specialize rules IR1 and 2 for our physical clocks as follows:

IR1′. For each i, if P_i does not receive a message at physical time t, then C_i is differentiable at t and $dC_i(t)/dt > 0$.

IR2′. (a) If P_i sends a message m at physical time t, then m contains a timestamp $T_m = C_i(t)$. (b) Upon receiving a message m at time t', process P_j sets $C_j(t')$ equal to maximum $(C_j(t' - 0), T_m + \mu_m)$.[9]

Although the rules are formally specified in terms of the physical time parameter, a process only needs to know its own clock reading and the timestamps of messages it receives. For mathematical convenience, we are assuming that each event occurs at a precise instant of physical time, and different events in the same process occur at different times. These rules are then specializations of rules IR1 and IR2, so our system of clocks satisfies the Clock Condition. The fact that real events have a finite duration causes no difficulty in implementing the algorithm. The only real concern in the implementation is making sure that the discrete clock ticks are frequent enough so C1 is maintained.

We now show that this clock synchronizing algorithm can be used to satisfy condition PC2. We assume that the system of processes is described by a directed graph in which an arc from process P_i to process P_j represents a communication line over which messages are sent directly from P_i to P_j. We say that a message is sent over this arc every τ seconds if for any t, P_i sends at least one message to P_j between physical times t and $t + \tau$. The *diameter* of the directed graph is the smallest number d such that for any pair of distinct processes P_j, P_k, there is a path from P_j to P_k having at most d arcs.

In addition to establishing PC2, the following theorem bounds the length of time it can take the clocks to become synchronized when the system is first started.

THEOREM. Assume a strongly connected graph of processes with diameter d which always obeys rules IR1′ and IR2′. Assume that for any message m, $\mu_m \leq \mu$ for some constant μ, and that for all $t \geq t_0$: (a) PC1 holds. (b) There are constants τ and ξ such that every τ seconds a message with an unpredictable delay less than ξ is sent over every arc. Then PC2 is satisfied with $\epsilon \approx d(2\kappa\tau + \xi)$ for all $t \gtrsim t_0 + \tau d$, where the approximations assume $\mu + \xi \ll \tau$.

The proof of this theorem is surprisingly difficult, and is given in the Appendix. There has been a great deal of work done on the problem of synchronizing physical clocks. We refer the reader to [4] for an intro-

[9] $C_j(t' - 0) = \lim_{\delta \to 0} C_j(t' - |\delta|)$.

duction to the subject. The methods described in the literature are useful for estimating the message delays μ_m and for adjusting the clock frequencies dC_i/dt (for clocks which permit such an adjustment). However, the requirement that clocks are never set backwards seems to distinguish our situation from ones previously studied, and we believe this theorem to be a new result.

Conclusion

We have seen that the concept of "happening before" defines an invariant partial ordering of the events in a distributed multiprocess system. We described an algorithm for extending that partial ordering to a somewhat arbitrary total ordering, and showed how this total ordering can be used to solve a simple synchronization problem. A future paper will show how this approach can be extended to solve any synchronization problem.

The total ordering defined by the algorithm is somewhat arbitrary. It can produce anomalous behavior if it disagrees with the ordering perceived by the system's users. This can be prevented by the use of properly synchronized physical clocks. Our theorem showed how closely the clocks can be synchronized.

In a distributed system, it is important to realize that the order in which events occur is only a partial ordering. We believe that this idea is useful in understanding any multiprocess system. It should help one to understand the basic problems of multiprocessing independently of the mechanisms used to solve them.

Appendix

Proof of the Theorem

For any i and t, let us define C_i^t to be a clock which is set equal to C_i at time t and runs at the same rate as C_i, but is never reset. In other words,

$$C_i^t(t') = C_i(t) + \int_t^{t'} [dC_i(t)/dt]dt \tag{1}$$

for all $t' \geq t$. Note that

$$C_i(t') \geq C_i^t(t') \text{ for all } t' \geq t. \tag{2}$$

Suppose process P_1 at time t_1 sends a message to process P_2 which is received at time t_2 with an unpredictable delay $\leq \xi$, where $t_0 \leq t_1 \leq t_2$. Then for all $t \geq t_2$ we have:

$$\begin{aligned} C_2^{t_2}(t) &\geq C_2^{t_2}(t_2) + (1-\kappa)(t - t_2) && \text{[by (1) and PC1]} \\ &\geq C_1(t_1) + \mu_m + (1-\kappa)(t - t_2) && \text{[by IR2' (b)]} \\ &= C_1(t_1) + (1-\kappa)(t - t_1) - [(t_2 - t_1) - \mu_m] + \kappa(t_2 - t_1) \\ &\geq C_1(t_1) + (1-\kappa)(t - t_1) - \xi. \end{aligned}$$

Hence, with these assumptions, for all $t \geq t_2$ we have:

$$C_2^{t_2}(t) \geq C_1(t_1) + (1-\kappa)(t - t_1) - \xi. \tag{3}$$

Now suppose that for $i = 1, \ldots, n$ we have $t_i \leq t_i' < t_{i+1}$, $t_0 \leq t_1$, and that at time t_i' process P_i sends a message to process P_{i+1} which is received at time t_{i+1} with an unpredictable delay less than ξ. Then repeated application of the inequality (3) yields the following result for $t \geq t_{n+1}$.

$$C_{n+1}^{t_{n+1}}(t) \geq C_1(t_1') + (1-\kappa)(t - t_1') - n\xi. \tag{4}$$

From PC1, IR1' and 2' we deduce that

$$C_1(t_1') \geq C_1(t_1) + (1-\kappa)(t_1' - t_1).$$

Combining this with (4) and using (2), we get

$$C_{n+1}(t) \geq C_1(t_1) + (1-\kappa)(t - t_1) - n\xi \tag{5}$$

for $t \geq t_{n+1}$.

For any two processes P and P', we can find a sequence of processes $P = P_0, P_1, \ldots, P_{n+1} = P'$, $n \leq d$, with communication arcs from each P_i to P_{i+1}. By hypothesis (b) we can find times t_i, t_i' with $t_i' - t_i \leq \tau$ and $t_{i+1} - t_i' \leq \nu$, where $\nu = \mu + \xi$. Hence, an inequality of the form (5) holds with $n \leq d$ whenever $t \geq t_1 + d(\tau + \nu)$. For any i, j and any t, t_1 with $t_1 \geq t_0$ and $t \geq t_1 + d(\tau + \nu)$ we therefore have:

$$C_i(t) \geq C_j(t_1) + (1-\kappa)(t - t_1) - d\xi. \tag{6}$$

Now let m be any message timestamped T_m, and suppose it is sent at time t and received at time t'. We pretend that m has a clock C_m which runs at a constant rate such that $C_m(t) = t_m$ and $C_m(t') = t_m + \mu_m$. Then $\mu_m \leq t' - t$ implies that $dC_m/dt \leq 1$. Rule IR2' (b) simply sets $C_j(t')$ to *maximum* $(C_j(t' - 0), C_m(t'))$. Hence, clocks are reset only by setting them equal to other clocks.

For any time $t_x \geq t_0 + \mu/(1-\kappa)$, let C_x be the clock having the largest value at time t_x. Since all clocks run at a rate less than $1 + \kappa$, we have for all i and all $t \geq t_x$:

$$C_i(t) \leq C_x(t_x) + (1+\kappa)(t - t_x). \tag{7}$$

We now consider the following two cases: (i) C_x is the clock C_q of process P_q. (ii) C_x is the clock C_m of a message sent at time t_1 by process P_q. In case (i), (7) simply becomes

$$C_i(t) \leq C_q(t_x) + (1+\kappa)(t - t_x). \tag{8i}$$

In case (ii), since $C_m(t_1) = C_q(t_1)$ and $dC_m/dt \leq 1$, we have

$$C_x(t_x) \leq C_q(t_1) + (t_x - t_1).$$

Hence, (7) yields

$$C_i(t) \leq C_q(t_1) + (1+\kappa)(t - t_1). \tag{8ii}$$

Since $t_x \geq t_0 + \mu/(1-\kappa)$, we get

$$\begin{aligned} C_q(t_x - \mu/(1-\kappa)) &\leq C_q(t_x) - \mu && \text{[by PC1]} \\ &\leq C_m(t_x) - \mu && \text{[by choice of } m\text{]} \\ &\leq C_m(t_x) - (t_x - t_1)\mu_m/\nu_m && [\mu_m \leq \mu,\ t_x - t_1 \leq \nu_m] \\ &= T_m && \text{[by definition of } C_m\text{]} \\ &= C_q(t_1) && \text{[by IR2'(a)].} \end{aligned}$$

Hence, $C_q(t_x - \mu/(1-\kappa)) \leq C_q(t_1)$, so $t_x - t_1 \leq \mu/(1-\kappa)$ and thus $t_1 \geq t_0$.

Letting $t_1 = t_x$ in case (i), we can combine (8i) and (8ii) to deduce that for any t, t_x with $t \geq t_x \geq t_0 + \mu/(1 - \kappa)$ there is a process P_q and a time t_1 with $t_x - \mu/(1 - \kappa) \leq t_1 \leq t_x$ such that for all i:

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \tag{9}$$

Choosing t and t_x with $t \geq t_x + d(\tau + \nu)$, we can combine (6) and (9) to conclude that there exists a t_1 and a process P_q such that for all i:

$$\begin{aligned} C_q(t_1) + (1 - \kappa)(t - t_1) - d\xi &\leq C_i(t) \\ &\leq C_q(t_1) + (1 + \kappa)(t - t_1) \end{aligned} \tag{10}$$

Letting $t = t_x + d(\tau + \nu)$, we get

$$d(\tau + \nu) \leq t - t_1 \leq d(\tau + \nu) + \mu/(1 - \kappa).$$

Combining this with (10), we get

$$\begin{aligned} C_q(t_1) + (t - t_1) - \kappa d(\tau + \nu) - d\xi \leq C_i(t) \leq C_q(t_1) \\ + (t - t_1) + \kappa[d(\tau + \nu) + \mu/(1 - \kappa)] \end{aligned} \tag{11}$$

Using the hypotheses that $\kappa \ll 1$ and $\mu \leq \nu \ll \tau$, we can rewrite (11) as the following approximate inequality.

$$\begin{aligned} C_q(t_1) + (t - t_1) - d(\kappa\tau + \xi) &\lesssim C_i(t) \\ &\lesssim C_q(t_1) + (t - t_1) + d\kappa\tau. \end{aligned} \tag{12}$$

Since this holds for all i, we get

$$|C_i(t) - C_j(t)| \lesssim d(2\kappa\tau + \xi),$$

and this holds for all $t \gtrsim t_0 + d\tau$. □

Note that relation (11) of the proof yields an exact upper bound for $|C_i(t) - C_j(t)|$ in case the assumption $\mu + \xi \ll \tau$ is invalid. An examination of the proof suggests a simple method for rapidly initializing the clocks, or resynchronizing them if they should go out of synchrony for any reason. Each process sends a message which is relayed to every other process. The procedure can be initiated by any process, and requires less than $2d(\mu + \xi)$ seconds to effect the synchronization, assuming each of the messages has an unpredictable delay less than ξ.

Acknowledgment. The use of timestamps to order operations, and the concept of anomalous behavior are due to Paul Johnson and Robert Thomas.

Received March 1976; revised October 1977

References

1. Schwartz, J.T. *Relativity in Illustrations.* New York U. Press, New York, 1962.
2. Taylor, E.F., and Wheeler, J.A. *Space-Time Physics*, W.H. Freeman, San Francisco, 1966.
3. Lamport, L. The implementation of reliable distributed multiprocess systems. To appear in *Computer Networks.*
4. Ellingson, C., and Kulpinski, R.J. Dissemination of system-time. *IEEE Trans. Comm. Com-23*, 5 (May 1973), 605–624.

Programming Languages

J. J. Horning
Editor

Shallow Binding in Lisp 1.5

Henry G. Baker, Jr.
Massachusetts Institute of Technology

Shallow binding is a scheme which allows the value of a variable to be accessed in a bounded amount of computation. An elegant model for shallow binding in Lisp 1.5 is presented in which context-switching is an environment tree transformation called rerooting. Rerooting is completely general and reversible, and is optional in the sense that a Lisp 1.5 interpreter will operate correctly whether or not rerooting is invoked on every context change. Since rerooting leaves assoc [*v*, *a*] invariant, for all variables *v* and all environments *a*, the programmer can have access to a rerooting primitive, shallow[], which gives him dynamic control over whether accesses are shallow or deep, and which affects only the speed of execution of a program, not its semantics. In addition, multiple processes can be active in the same environment structure, so long as rerooting is an indivisible operation. Finally, the concept of rerooting is shown to combine the concept of shallow binding in Lisp with Dijkstra's display for Algol and hence is a general model for shallow binding.

Key Words and Phrases: Lisp 1.5, environment trees, FUNARG's, shallow binding, deep binding, multiprogramming, Algol display

CR Categories: 4.13, 4.22, 4.32

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0522.

Author's present address: Computer Science Department, University of Rochester, Rochester, NY 14627.

© 1978 ACM 0001-0782/78/0700-0565 $00.75

Chapter 2

Distributed Snapshots: Determining Global States of Distributed Systems

Authors:
K. Mani Chandy, Leslie Lamport

Publication year: 1985

Distributed Snapshots: Determining Global States of Distributed Systems

K. MANI CHANDY
University of Texas at Austin
and
LESLIE LAMPORT
Stanford Research Institute

This paper presents an algorithm by which a process in a distributed system determines a global state of the system during a computation. Many problems in distributed systems can be cast in terms of the problem of detecting global states. For instance, the global state detection algorithm helps to solve an important class of problems: stable property detection. A stable property is one that persists: once a stable property becomes true it remains true thereafter. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared." The stable property detection problem is that of devising algorithms to detect a given stable property. Global state detection can also be used for checkpointing.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; *network operating systems*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *multiprocessing/multiprogramming*; *mutual exclusion*; *scheduling*; *synchronization*; D.4.5 [**Operating Systems**]: Reliability—*backup procedures*; *checkpoint/restart*; *fault-tolerance*; *verification*

General Terms: Algorithms

Additional Key Words and Phrases: Global States, Distributed deadlock detection, distributed systems, message communication systems

1. INTRODUCTION

This paper presents algorithms by which a process in a distributed system can determine a global state of the system during a computation. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and the messages it sends and receives; *it can record nothing else.* To determine a global system state, a process p must enlist the

This work was supported in part by the Air Force Office of Scientific Research under Grant AFOSR 81-0205 and in part by the National Science Foundation under Grant MCS 81-04459.
Authors' addresses: K. M. Chandy, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; L. Lamport, Stanford Research Institute, Menlo Park, CA 94025.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1985 ACM 0734-2071/85/0200-0063 $00.75

cooperation of other processes that must record their own local states and send the recorded local states to p. All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state. The global-state-detection algorithm is to be superimposed on the underlying computation: it must run concurrently with, but not alter, this underlying computation.

The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds—a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define "meaningful" and then to determine how the photographs should be taken.

We now describe an important class of problems that can be solved with the global-state-detection algorithm. Let y be a predicate function defined on the global states of a distributed system D; that is, $y(S)$ is true or false for a global state S of D. The predicate y is said to be a *stable property* of D if $y(S)$ implies $y(S')$ for all global states S' of D reachable from global state S of D. In other words, if y is a stable property and y is true at a point in a computation of D, then y is true at all later points in that computation. Examples of stable properties are "computation has terminated," "the system is deadlocked," and "all tokens in a token ring have disappeared."

Several distributed-system problems can be formulated as the general problem of devising an algorithm by which a process in a distributed system can determine whether a stable property y of the system holds. Deadlock detection [2, 5, 8, 9, 11] and termination detection [1, 4, 10] are special cases of the stable-property detection problem. Details of the algorithm are presented later. The basic idea of the algorithm is that a global state S of the system is determined and $y(S)$ is computed to see if the stable property y holds.

Several algorithms for solving deadlock and termination problems by determining the global states of distributed systems have been published. Gligor and Shattuck [5] state that many of the published algorithms are incorrect and impractical. A reason for the incorrect or impractical algorithms may be that the relationships among local process states, global system states, and points in a distributed computation are not well understood. One of the contributions of this paper is to define these relationships.

Many distributed algorithms are structured as a sequence of phases, where each phase consists of a transient part in which useful work is done, followed by a stable part in which the system cycles endlessly and uselessly. The presence of stable behavior indicates the end of a phase. A phase is similar to a series of

iterations in a sequential program, which are repeated until successive iterations produce no change, that is, stability is attained. Stability must be detected so that one phase can be terminated and the next phase initiated [10]. The termination of a computational phase is not identical to the termination of a computation. When a computation terminates, all activities cease—messages are not sent and process states do not change. There may be activity during the stable behavior that indicates the end of a computational phase—messages may be sent and received, and processes may change state, but this activity serves no purpose other than to signal the end of a phase. In this paper, we are concerned with the detection of stable system properties; the cessation of activity is only one example of a stable property.

Strictly speaking, properties such as "the system is deadlocked" are not stable if the deadlock is "broken" and computation is reinitiated. However, to keep exposition simple, we shall partition the overall problem into the problems of (1) detecting the termination of one phase (and informing all processes that a phase has ended) and (2) initiating a new phase. The following is a stable property: "the kth computational phase has terminated," $k = 1, 2, \ldots$. Hence, the methods presented in this paper are applicable to detecting the termination of the kth phase for a given k.

In this paper we restrict attention to the problem of detecting stable properties. The problem of initiating the next phase of computation is not considered here because the solution to that problem varies significantly depending on the application, being different for database deadlock detection than for detecting the termination of a diffusing computation.

We have to present our algorithms in terms of a model of a system. The model chosen is not important in itself; we could have couched our discussion in terms of other models. We shall describe our model informally and only to the level of detail necessary to make the algorithms clear.

2. MODEL OF A DISTRIBUTED SYSTEM

A distributed system consists of a finite set of processes and a finite set of channels. It is described by a labeled, directed graph in which the vertices represent processes and the edges represent channels. Figure 1 is an example.

Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent. (The infinite buffer assumption is made for ease of exposition: bounded buffers may be assumed provided there exists a proof that no process attempts to add a message to a full buffer.) The delay experienced by a message in a channel is arbitrary but finite. The sequence of messages received along a channel is an initial subsequence of the sequence of messages sent along the channel. The *state* of a channel is the sequence of messages sent along the channel, excluding the messages received along the channel.

A process is defined by a set of states, an initial state (from this set), and a set of events. An event e in a process p is an atomic action that may change the state of p itself and the state of *at most one* channel c incident on p: the state of c may be changed by the sending of a message along c (if c is directed away from p) or the receipt of a message along c (if c is directed towards p). An event e is defined by (1) the process p in which the event occurs, (2) the state s of p immediately

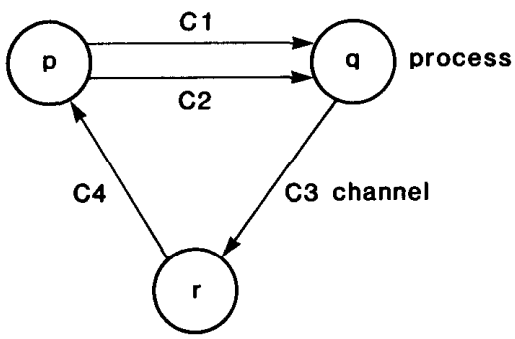


Fig. 1. A distributed system with processes p, q, and r and channels c1, c2, c3, and c4.

before the event, (3) the state s' of p immediately after the event, (4) the channel c (if any) whose state is altered by the event, and (5) the message M, if any, sent along c (if c is a channel directed away from p) or received along c (if c is directed towards p). We define e by the 5-tuple $\langle p, s, s', M, c\rangle$, where M and c are a special symbol, *null*, if the occurrence of e does not change the state of any channel.

A global state of a distributed system is a set of component process and channel states: the *initial* global state is one in which the state of each process is its initial state and the state of each channel is the empty sequence. The occurrence of an event may change the global state. Let $e = \langle p, s, s', M, c\rangle$ we say *e can occur in* global state S if and only if (1) the state of process p in global state S is s and (2) if c is a channel directed towards p, then the state of c in global state S is a sequence of messages with M at its head. We define a function *next*, where $next(S, e)$ is the global state immediately after the occurrence of event e in global state S. The value of $next(S, e)$ is defined only if event e can occur in global state S, in which case $next(S, e)$ is the global state identical to S except that: (1) the state of p in $next(S, e)$ is s'; (2) if e is a channel directed towards p, then the state of c in $next(S, e)$ is c's state in S with message M deleted from its head; and (3) if c is a channel directed away from p, then the state of c in $next(S, e)$ is the same as c's state in S with message M added to the tail.

Let $seq = (e_i\colon 0 \le i \le n)$ be a sequence of events in component processes of a distributed system. We say that *seq* is a *computation of the system* if and only if event e_i can occur in global state S_i, $0 \le i \le n$, where S_0 is the initial global state and

$$S_{i+1} = next(S_i, e_i) \qquad \text{for} \quad 0 \le i \le n.$$

An alternate model, based on Lamport [6], which views computations as partially ordered sets of events, is given in [7].

Example 2.1. To illustrate the definition of a distributed system, consider a simple system consisting of two processes p and q, and two channels c and c' as shown in Figure 2.

The system contains one *token* that is passed from one process to another, and hence we call this system the "single-token conservation" system. Each process has two states, s_0 and s_1, where s_0 is the state in which the process does not possess the token and s_1 is the state in which it does. The initial state of p is s_1 and of q is s_0. Each process has two events: (1) a transition from s_1 to s_0 with the

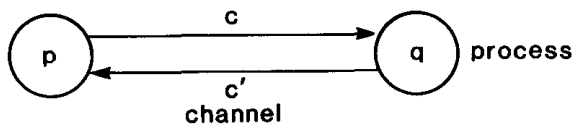


Fig. 2. The simple distributed system of Examples 2.1 and 2.2.

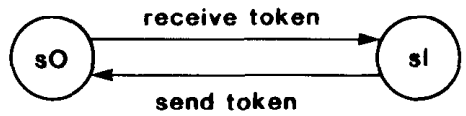


Fig. 3. State-transition diagram of a process in Example 2.1.

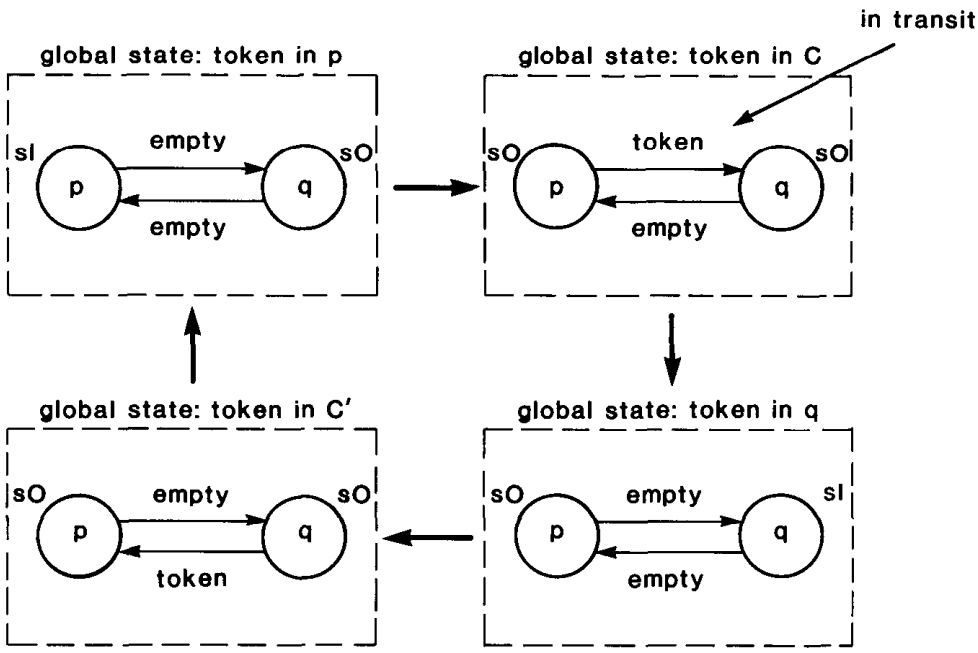


Fig. 4. Global states and transitions of the single-token conservation system.

sending of the token, and (2) a transition from s_0 to s_1 with the receipt of the token. The state-transition diagram for a process is shown in Figure 3. The global states and transitions are shown in Figure 4.

A system computation corresponds to a path in the global-state-transition diagram (Figure 4) starting at the initial global state. Examples of system computations are: (1) the empty sequence and (2) ⟨p sends token, q receives token, q sends token⟩. The following sequence is not a computation of the system: ⟨p sends token, q sends token⟩, because the event "q sends token" cannot occur while q is in the state s_0.

For brevity, the four global states, in order of transition (see Figure 4), will be called (1) in-p, (2) in-c, (3) in-q, and (4) in-c', to denote the location of the token. This example will be used later to motivate the algorithm. □

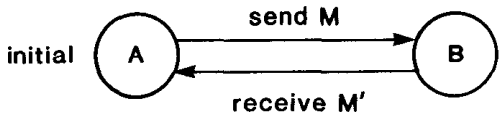


Fig. 5. State-transition diagram for process p in Example 2.2.

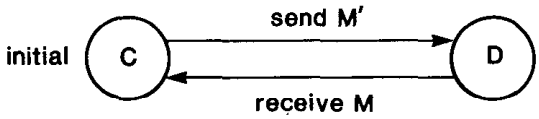


Fig. 6. State-transition diagram for process q in Example 2.2.

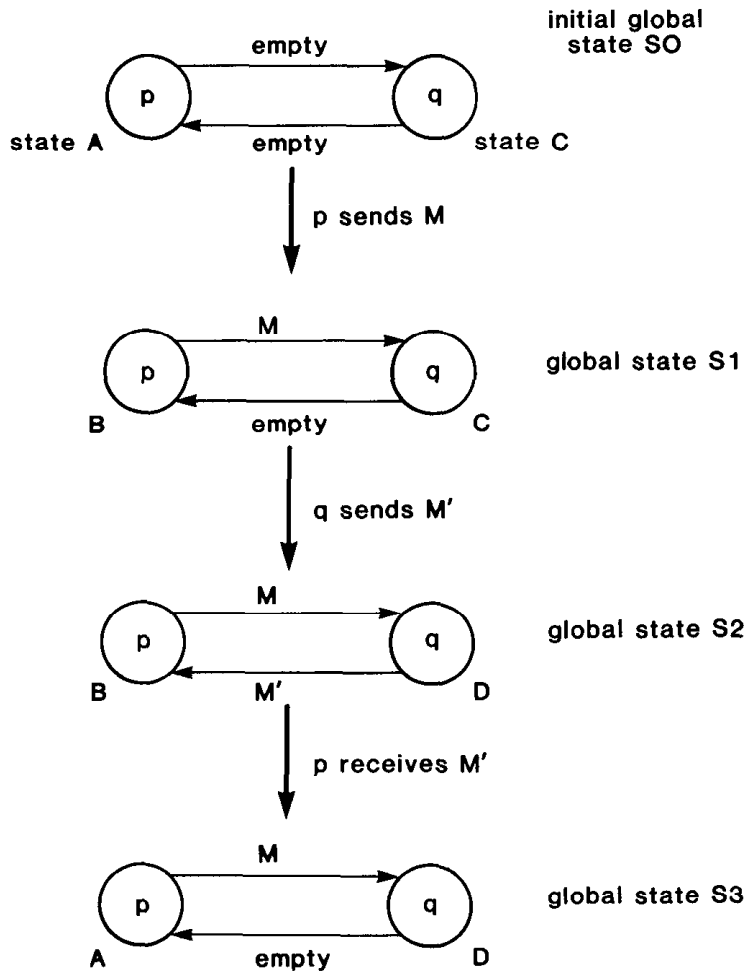


Fig. 7. A computation for Example 2.2.

Example 2.2. This example illustrates nondeterministic computations. Nondeterminism plays an interesting role in the snapshot algorithm.

In Example 2.1 there is exactly one event possible in each global state. Consider a system with the same topology as Example 2.1 (see Figure 2) but where the processes p and q are defined by the state-transition diagrams of Figures 5 and 6.

An example of a computation is shown in Figure 7. The reader should observe that there may be more than one transition allowable from a global state. For instance, events "p sends M" and "q sends M'" may occur in the initial global state, and the next states after these events are different. □

3. THE ALGORITHM

3.1. Motivation for the Steps of the Algorithm

The global-state recording algorithm works as follows: Each process records its own state, and the two processes that a channel is incident on cooperate in recording the channel state. We cannot ensure that the states of all processes and channels will be recorded at the same instant because there is no global clock; however, we require that the recorded process and channel states form a "meaningful" global system state.

The global-state recording algorithm is to be superimposed on the underlying computation, that is, it must run concurrently with, but not alter, the underlying computation. The algorithm may send messages and require processes to carry out computations; however, the messages and computation required to record the global state must not interfere with the underlying computation.

We now consider an example to motivate the steps of the algorithm. In the example we shall assume that we can record the state of a channel instantaneously; we postpone discussion of how the channel state is recorded. Let c be a channel from p to q. The purpose of the example is to gain an intuitive understanding of the relationship between the instant at which the state of channel c is to be recorded and the instants at which the states of processes p and q are to be recorded.

Example 3.1. Consider the single-token conservation system. Assume that the state of process p is recorded in global state in-p. Then the state recorded for p shows the token in p. Now assume that the global state transits to in-c (because p sends the token). Suppose the states of channels c and c' and of process q were recorded in global state in-c, so the state recorded for channel c shows it with the token and the states recorded for channel c' and process q show them not in possession of the token. The composite global state recorded in this fashion would show two tokens in the system, one in p and the other in c. But a global state with two tokens is unreachable from the initial global state in a *single-token* conservation system! The inconsistency arises because the state of p is recorded *before* p sent a message along c and the state of c is recorded *after* p sent the message. Let n be the number of messages sent along c before p's state is recorded, and let n' be the number of messages sent along c before c's state is recorded. Our example suggests that the recorded global state may be inconsistent if $n < n'$.

Now consider an alternate scenario. Suppose the state of c is recorded in global state in-p, the system then transits to global state in-c, and the states of c', p, and q are recorded in global state in-c. The recorded global state shows *no* tokens in the system. This example suggests that the recorded global state may be inconsistent if the state of c is recorded *before* p sends a message along c and the state of p is recorded *after* p sends a message along c, that is, if $n > n'$. □

We learn from these examples that (in general) a consistent global state requires

$$n = n'. \tag{1}$$

Let m be the number of messages received along c before q's state is recorded. Let m' be the number of messages received along c before c's state is recorded. We leave it up to the reader to extend the example to show that consistency requires

$$m = m'. \tag{2}$$

In every state, the number of messages received along a channel cannot exceed the number of messages sent along that channel, that is,

$$n' \geq m'. \tag{3}$$

From the above equations,

$$n \geq m. \tag{4}$$

The state of channel c that is recorded must be the sequence of messages sent along the channel before the sender's state is recorded, excluding the sequence of messages received along the channel before the receiver's state is recorded—that is, if $n' = m'$, the recorded state of c must be the empty sequence, and if $n' > m'$, the recorded state of c must be the $(m' + 1)$st, . . . , n'th messages sent by p along c. This fact and eqs. (1)–(4) suggest a simple algorithm by which q can record the state of channel c. Process p sends a special message, called a *marker*, after the nth message it sends along c (and before sending further messages along c). The marker has no effect on the underlying computation. The state of c is the sequence of messages received by q after q records its own state and before q receives the marker along c. To ensure eq. (4), q must record its state, if it has not done so already, after receiving a marker along c and before q receives further messages along c.

Our example suggests the following outline for a global state detection algorithm.

3.2 Global-State-Detection Algorithm Outline

Marker-Sending Rule for a Process p. For each channel c, incident on, and directed away from p:

p sends one marker along c after p records its state and before p sends further messages along c.

Marker-Receiving Rule for a Process q. On receiving a marker along a channel c:

if q has not recorded its state **then**
  begin q records its state;
    q records the state c as the empty sequence
  end
else q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

3.3 Termination of the Algorithm

The marker receiving and sending rules guarantee that if a marker is received along every channel, then each process will record its state and the states of all

incoming channels. To ensure that the global-state recording algorithm terminates in finite time, each process must ensure that (L1) no marker remains forever in an incident input channel and (L2) it records its state within finite time of initiation of the algorithm.

The algorithm can be initiated by one or more processes, each of which records its state spontaneously, without receiving markers from other processes; we postpone discussion of what may cause a process to record its state spontaneously. If process p records its state and there is a channel from p to a process q, then q will record its state in finite time because p will send a marker along the channel and q will receive the marker in finite time (L1). Hence if p records its state and there is a path (in the graph representing the system) from p to a process q, then q will record its state in finite time because, by induction, every process along the path will record its state in finite time. Termination in finite time is ensured if for every process q: q spontaneously records its state or there is a path from a process p, which spontaneously records its state, to q.

In particular, if the graph is strongly connected and at least one process spontaneously records its state, then all processes will record their states in finite time (provided L1 is ensured).

The algorithm described so far allows each process to record its state and the states of incoming channels. The recorded process and channel states must be collected and assembled to form the recorded global state. We shall not describe algorithms for collecting the recorded information because such algorithms have been described elsewhere [4, 10]. A simple algorithm for collecting information in a system whose topology is strongly connected is for each process to send the information it records along all outgoing channels, and for each process receiving information for the first time to copy it and propagate it along all of its outgoing channels. All the recorded information will then get to all the processes in finite time, allowing all processes to determine the recorded global state.

4. PROPERTIES OF THE RECORDED GLOBAL STATE

To gain an intuitive understanding of the properties of the global state recorded by the algorithm, we shall study Example 2.2. Assume that the state of p is recorded in global state S_0 (Figure 7), so the state recorded for p is A. After recording its state, p sends a marker along channel c. Now assume that the system goes to global state S_1, then S_2, and then S_3 while the marker is still in transit, and the marker is received by q when the system is in global state S_3. On receiving the marker, q records its state, which is D, and records the state of c to be the empty sequence. After recording its state, q sends a marker along channel c'. On receiving the marker, p records the state of c' as the sequence consisting of the single message M'. The recorded global state S^* is shown in Figure 8. The recording algorithm was initiated in global state S_0 and terminated in global state S_3.

Observe that the global state S^* recorded by the algorithm is not identical to any of the global states S_0, S_1, S_2, S_3 that occurred in the computation. Of what use is the algorithm if the recorded global state never occurred? We shall now answer this question.

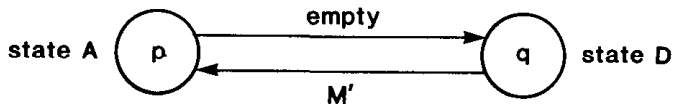


Fig. 8. A recorded global state for Example 2.2.

Let $seq = (e_i, 0 \leq i)$ be a distributed computation, and let S_i be the global state of the system immediately before event e_i, $0 \leq i$, in *seq*. Let the algorithm be initiated in global state S_ι and let it terminate in global state S_ϕ, $0 \leq \iota \leq \phi$; in other words, the algorithm is initiated after $e_{\iota-1}$ if $\iota > 0$, and before e_ι, and it terminates after $e_{\phi-1}$ if $\phi > 0$, and before e_ϕ. We observed in Example 2.2 that the recorded global state S^* may be different from all global states S_k, $\iota \leq k \leq \phi$.

We shall show that:

(1) S^* is reachable from S_ι, and
(2) S_ϕ is reachable from S^*.

Specifically, we shall show that there exists a computation *seq'* where

(1) *seq'* is a permutation of *seq*, such that S_ι, S^* and S_ϕ occur as global states in *seq'*,
(2) $S_\iota = S^*$ or S_ι occurs earlier than S^*, and
(3) $S_\phi = S^*$ or S^* occurs earlier than S_ϕ in *seq'*.

THEOREM 1. *There exists a computation* $seq' = (e_i', 0 \leq i)$ *where*

(1) *For all i, where* $i < \iota$ *or* $i \geq \phi$: $e_i' = e_i$, and
(2) *the subsequence* $(e_i', \iota \leq i < \phi)$ *is a permutation of the subsequence* $(e_i, \iota \leq i < \phi)$, *and*
(3) *for all i where* $i \leq \iota$ *or* $i \geq \phi$: $S_i' = S_i$, *and*
(4) *there exists some k,* $\iota \leq k \leq \phi$, *such that* $S^* = S_k'$.

PROOF. Event e_i in *seq* is called a *prerecording* event if and only if e_i is in a process p and p records its state *after* e_i in *seq*. Event e_i in *seq* is called a *postrecording* event if and only if it is not a prerecording event—that is, if e_i is in a process p and p records its state *before* e_i in *seq*. All events e_i, $i < \iota$, are prerecording events and all events e_i, $i \geq \phi$, are postrecording events in *seq*. There *may be* a postrecording event e_{j-1} before a prerecording event e_j for some j, $\iota < j < \phi$; this can occur only if e_{j-1} and e_j are in different processes (because if e_{j-1} and e_j are in the same process and e_{j-1} is a postrecording event, then so is e_j).

We shall derive a computation *seq'* by permuting *seq*, where all prerecording events occur before all postrecording events in *seq'*. We shall show that S^* is the global state in *seq'* after all prerecording events and before all postrecording events.

Assume that there is a postrecording event e_{j-1} before a prerecording event e_j in *seq*. We shall show that the sequence obtained by interchanging e_{j-1} and e_j must also be a computation. Events e_{j-1} and e_j must be on different processes. Let p be the process in which e_{j-1} occurs, and let q be the process in which e_j occurs. There cannot be a message sent at e_{j-1} which is received at e_j because (1)

if a message is sent along a channel c when event e_{j-1} occurs, then a marker must have been sent along c *before* e_{j-1}, since e_{j-1} is a postrecording event, and (2) if the message is received along channel c when e_j occurs, then the marker must have been received along c *before* e_j occurs (since channels are first-in-first-out), in which case (by the marker-receiving rule) e_j would be a postrecording event too.

The state of process q is not altered by the occurrence of event e_{j-1} because e_{j-1} is in a different process p. If e_j is an event in which q receives a message M along a channel c, then M must have been the message at the head of c before event e_{j-1}, since a message sent at e_{j-1} cannot be received at e_j. Hence event e_j can occur in global state S_{j-1}.

The state of process p is not altered by the occurrence of e_j. Hence e_{j-1} can occur after e_j. Hence the sequence of events $e_1, \ldots, e_{j-2}, e_j, e_{j-1}$ is a computation. From the arguments in the last paragraph it follows that the global state after computation $e_1, \ldots, e_j$ is the same as the global state after computation $e_1, \ldots, e_{j-2}, e_j, e_{j-1}$.

Let seq^* be a permutation of seq that is identical to seq except that e_j and e_{j-1} are interchanged. Then seq^* must also be a computation. Let $\bar{S}_i$ be the global state immediately before the ith event in seq^*. From the arguments of the previous paragraph,

$$\bar{S}_i = S_i \qquad \text{for all} \quad i \quad \text{where} \quad i \neq j.$$

By repeatedly swapping postrecording events that immediately follow prerecording events, we see that there exists a permutation seq' of seq in which

(1) all prerecording events precede all postrecording events,
(2) seq' is a computation,
(3) for all i where $i < \iota$ or $i \geq \phi$: $e_i' = e_i$, and
(4) for all i where $i \leq \iota$ or $i \geq \phi$: $S_i' = S_i$.

Now we shall show that the global state after all prerecording events and before all postrecording events in seq' is S^*. To do this, we need to show that

(1) the state of each process p in S^* is the same as its state after the process computation consisting of the sequence of prerecorded events on p, and
(2) the state of each channel c in S^* is (sequence of messages corresponding to prerecorded sends on c) − (sequence of messages corresponding to prerecorded receives on c).

The proof of the first part is trivial. Now we prove part (2). Let c be a channel from process p to process q. The state of channel c recorded in S^* is the sequence of messages received on c by q *after* q records its state *and before* q receives a marker on c. The sequence of messages sent by p along c before p sends a marker along c is the sequence corresponding to prerecorded sends on c. Part (2) now follows. □

Example 4.1. The purpose of this example is to show how the computation seq' is derived from the computation seq. Consider Example 2.2. The sequence

of events shown in the computation of Figure 7 is

e_0: p sends M and changes state to B (a postrecording event)

e_1: q sends M' and changes state to D (a prerecording event)

e_2: p receives M' and changes state to A (a postrecording event)

Since e_0, a postrecording event, immediately precedes e_1, a prerecording event, we interchange them, to get the permuted sequence seq':

e'_0: q sends M' and changes state to D (a prerecording event)

e'_1: p sends M and changes state to B (a postrecording event)

e'_2: p receives M' and changes state to A (a postrecording event)

In seq', all prerecording events precede all postrecording events. We leave it to the reader to show that the global state after e'_0 is the recorded global state.

5. STABILITY DETECTION

We now solve the stability-detection problem described in Section 1. We study the stability-detection problem because it is a paradigm for many practical problems, such as distributed deadlock detection.

A stability-detection algorithm is defined as follows:

Input: A stable property y

Output: A Boolean value *definite* with the property:
$(y(S_\iota) \rightarrow definite)$ and $(definite \rightarrow y(S_\phi))$

where S_ι and S_ϕ are the global states of the system when the algorithm is initiated and when it terminates, respectively. (The symbol $\rightarrow$ denotes logical implication.)

The input to the algorithm is (the definition of) function y. During the execution of the algorithm the value $y(S)$ for some global state S may be determined by a process in the system by applying the *externally defined* function y to global state S. By the output of the algorithm being a Boolean value *definite* we mean that (1) some specially designated process (say p) enters and thereafter remains in some special state to symbolize an output of *definite* = *true*, and (2) p enters and remains in some other special state to symbolize an output of *definite* = *false*.

Definite = *true* implies that the stable property holds when the algorithm *terminates*. However, *definite* = *false* implies that the stable property does not hold when the algorithm is *initiated*. We emphasize that *definite* = *true* gives us information about the state of the system at the *termination* of the algorithm, whereas *definite* = *false* gives us information about the system state at the *initiation* of the algorithm. In particular, we cannot deduce from *definite* = *false* that the stable property does not hold at termination of the algorithm.

The solution to the stability detection problem is

```
begin
    record a global state S*;
    definite := y(S*)
end.
```

The correctness of the stability detection algorithm follows from the following facts:

(1) S^* is reachable from S_ι,
(2) S_ϕ is reachable from S^* (Theorem 1), and
(3) $y(S) \rightarrow y(S')$ for all S' reachable from S (definition of a stable property).

ACKNOWLEDGMENTS

J. Misra's contributions in defining the problem of global state detection are gratefully acknowledged. We are grateful to E. W. Dijkstra and C. S. Scholten for their comments—particularly regarding the proof of Theorem 1. The outline of the current version of the proof was suggested by them. Dijkstra's note [3] on the subject provides colorful insight into the problem of stability detection. Thanks are due to C. A. R. Hoare, F. Schneider, and G. Andrews who helped us with detailed comments. We are grateful to Anita Jones and anonymous referees for suggestions.

REFERENCES

1. CHANDY, K. M., AND MISRA, J. Distributed computation on graphs: Shortest path algorithms. *Commun. ACM 25,* 11 (Nov. 1982), 833–837.
2. CHANDY, K. M., MISRA, J., AND HAAS, L. Distributed deadlock detection. *ACM Trans. Comput. Syst.* 1, 2 (May 1983), 144–156.
3. DIJKSTRA, E. W. The distributed snapshot of K. M. Chandy and L. Lamport. Tech. Rep. EWD 864a, Univ. of Texas, Austin, Tex., 1984.
4. DIJKSTRA, E. W., AND SCHOLTEN, C. S. Termination detection for diffusing computations. *Inf. Proc. Lett. 11,* 1 (Aug. 1980), 1–4.
5. GLIGOR, V. D., AND SHATTUCK, S. H. Deadlock detection in distributed systems. *IEEE Trans. Softw. Eng. SE-6,* 5 (Sep. 1980), 435–440.
6. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (Jul. 1978), 558–565.
7. LAMPORT, L., AND CHANDY, K. M. On partially-ordered event models of distributed computations. Submitted for publication.
8. MAHOUD, S. A., AND RIORDAN, J. S. Software controlled access to distributed databases. *INFOR 15,* 1 (Feb. 1977), 22–36.
9. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed data bases. *IEEE Trans. Softw. Eng. SE-5,* 3 (May 1979), 195–202.
10. MISRA, J., AND CHANDY, K. M. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 37–43.
11. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7,* 2 (Jun. 1982), 187–208.

Received January 1984; revised September 1984; accepted 7 December 1984

Chapter 3

The Byzantine Generals Problem

Authors:
Leslie Lamport, Robert Shostak, Marshall Pease

Publication year: 1982

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

1. INTRODUCTION

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked—namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem. We devote the major part of the paper to a discussion of this abstract problem and conclude by indicating how our solutions can be used in implementing a reliable computer system.

We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals

This research was supported in part by the National Aeronautics and Space Administration under contract NAS1-15428 Mod. 3, the Ballistic Missile Defense Systems Command under contract DASG60-78-C-0046, and the Army Research Office under contract DAAG29-79-C-0102.
Authors' address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0164-0925/82/0700-0382 $00.75

may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that

A. All loyal generals decide upon the same plan of action.

The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition A regardless of what the traitors do.

The loyal generals should not only reach agreement, but should agree upon a reasonable plan. We therefore also want to insure that

B. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Condition B is hard to formalize, since it requires saying precisely what a bad plan is, and we do not attempt to do so. Instead, we consider how the generals reach a decision. Each general observes the enemy and communicates his observations to the others. Let $v(i)$ be the information communicated by the ith general. Each general uses some method for combining the values $v(1), \ldots, v(n)$ into a single plan of action, where n is the number of generals. Condition A is achieved by having all generals use the same method for combining the information, and Condition B is achieved by using a robust method. For example, if the only decision to be made is whether to attack or retreat, then $v(i)$ can be General i's opinion of which option is best, and the final decision can be based upon a majority vote among them. A small number of traitors can affect the decision only if the loyal generals were almost equally divided between the two possibilities, in which case neither decision could be called bad.

While this approach may not be the only way to satisfy conditions A and B, it is the only one we know of. It assumes a method by which the generals communicate their values $v(i)$ to one another. The obvious method is for the ith general to send $v(i)$ by messenger to each other general. However, this does not work, because satisfying condition A requires that every loyal general obtain the same values $v(1), \ldots, v(n)$, and a traitorous general may send different values to different generals. For condition A to be satisfied, the following must be true:

1. Every loyal general must obtain the same information $v(1), \ldots, v(n)$.

Condition 1 implies that a general cannot necessarily use a value of $v(i)$ obtained directly from the ith general, since a traitorous ith general may send different values to different generals. This means that unless we are careful, in meeting condition 1 we might introduce the possibility that the generals use a value of $v(i)$ different from the one sent by the ith general—even though the ith general is loyal. We must not allow this to happen if condition B is to be met. For example, we cannot permit a few traitors to cause the loyal generals to base their decision upon the values "retreat", . . . , "retreat" if every loyal general sent the value "attack". We therefore have the following requirement for each i:

2. If the ith general is loyal, then the value that he sends must be used by every loyal general as the value of $v(i)$.

We can rewrite condition 1 as the condition that for every i (whether or not the ith general is loyal),

1′. Any two loyal generals use the same value of $v(i)$.

Conditions 1′ and 2 are both conditions on the single value sent by the ith general. We can therefore restrict our consideration to the problem of how a single general sends his value to the others. We phrase this in terms of a commanding general sending an order to his lieutenants, obtaining the following problem.

Byzantine Generals Problem. A commanding general must send an order to his $n - 1$ lieutenant generals such that

IC1. All loyal lieutenants obey the same order.
IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Conditions IC1 and IC2 are called the *interactive consistency* conditions. Note that if the commander is loyal, then IC1 follows from IC2. However, the commander need not be loyal.

To solve our original problem, the ith general sends his value of $v(i)$ by using a solution to the Byzantine Generals Problem to send the order "use $v(i)$ as my value", with the other generals acting as the lieutenants.

2. IMPOSSIBILITY RESULTS

The Byzantine Generals Problem seems deceptively simple. Its difficulty is indicated by the surprising fact that if the generals can send only oral messages, then no solution will work unless more than two-thirds of the generals are loyal. In particular, with only three generals, no solution can work in the presence of a single traitor. An oral message is one whose contents are completely under the control of the sender, so a traitorous sender can transmit any possible message. Such a message corresponds to the type of message that computers normally send to one another. In Section 4 we consider signed, written messages, for which this is not true.

We now show that with oral messages no solution for three generals can handle a single traitor. For simplicity, we consider the case in which the only possible decisions are "attack" or "retreat". Let us first examine the scenario pictured in Figure 1 in which the commander is loyal and sends an "attack" order, but Lieutenant 2 is a traitor and reports to Lieutenant 1 that he received a "retreat" order. For IC2 to be satisfied, Lieutenant 1 must obey the order to attack.

Now consider another scenario, shown in Figure 2, in which the commander is a traitor and sends an "attack" order to Lieutenant 1 and a "retreat" order to Lieutenant 2. Lieutenant 1 does not know who the traitor is, and he cannot tell what message the commander actually sent to Lieutenant 2. Hence, the scenarios in these two pictures appear exactly the same to Lieutenant 1. If the traitor lies consistently, then there is no way for Lieutenant 1 to distinguish between these two situations, so he must obey the "attack" order in both of them. Hence, whenever Lieutenant 1 receives an "attack" order from the commander, he must obey it.

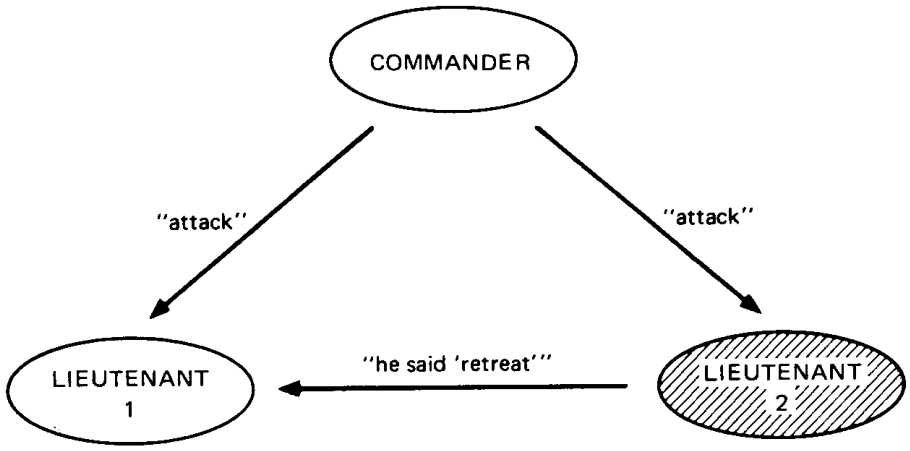


Fig. 1. Lieutenant 2 a traitor.

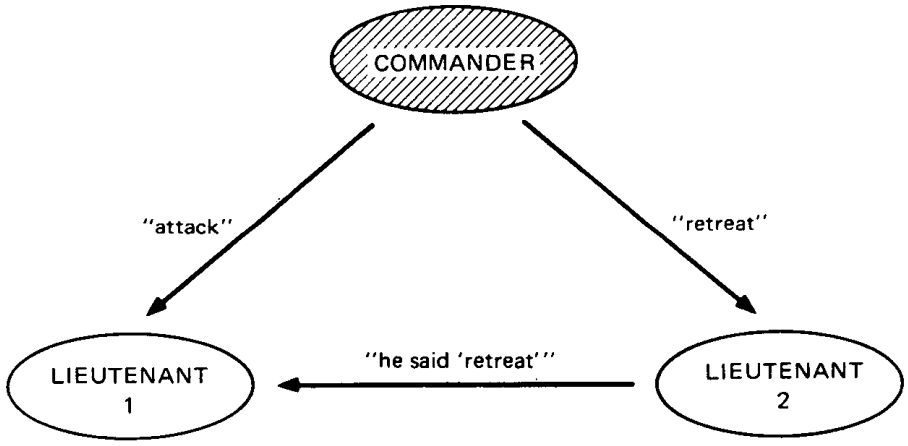


Fig. 2. The commander a traitor.

However, a similar argument shows that if Lieutenant 2 receives a "retreat" order from the commander then he must obey it even if Lieutenant 1 tells him that the commander said "attack". Therefore, in the scenario of Figure 2, Lieutenant 2 must obey the "retreat" order while Lieutenant 1 obeys the "attack" order, thereby violating condition IC1. Hence, no solution exists for three generals that works in the presence of a single traitor.

This argument may appear convincing, but we strongly advise the reader to be very suspicious of such nonrigorous reasoning. Although this result is indeed correct, we have seen equally plausible "proofs" of invalid results. We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm. For a rigorous proof of the impossibility of a three-general solution that can handle a single traitor, we refer the reader to [3].

Using this result, we can show that no solution with fewer than $3m + 1$ generals can cope with m traitors.[1] The proof is by contradiction—we assume such a

[1] More precisely, no such solution exists for three or more generals, since the problem is trivial for two generals.

solution for a group of $3m$ or fewer and use it to construct a three-general solution to the Byzantine Generals Problem that works with one traitor, which we know to be impossible. To avoid confusion between the two algorithms, we call the generals of the assumed solution Albanian generals, and those of the constructed solution Byzantine generals. Thus, starting from an algorithm that allows $3m$ or fewer Albanian generals to cope with m traitors, we construct a solution that allows three Byzantine generals to handle a single traitor.

The three-general solution is obtained by having each of the Byzantine generals simulate approximately one-third of the Albanian generals, so that each Byzantine general is simulating at most m Albanian generals. The Byzantine commander simulates the Albanian commander plus at most $m - 1$ Albanian lieutenants, and each of the two Byzantine lieutenants simulates at most m Albanian lieutenants. Since only one Byzantine general can be a traitor, and he simulates at most m Albanians, at most m of the Albanian generals are traitors. Hence, the assumed solution guarantees that IC1 and IC2 hold for the Albanian generals. By IC1, all the Albanian lieutenants being simulated by a loyal Byzantine lieutenant obey the same order, which is the order he is to obey. It is easy to check that conditions IC1 and IC2 of the Albanian generals solution imply the corresponding conditions for the Byzantine generals, so we have constructed the required impossible solution.

One might think that the difficulty in solving the Byzantine Generals Problem stems from the requirement of reaching exact agreement. We now demonstrate that this is not the case by showing that reaching approximate agreement is just as hard as reaching exact agreement. Let us assume that instead of trying to agree on a precise battle plan, the generals must agree only upon an approximate time of attack. More precisely, we assume that the commander orders the time of the attack, and we require the following two conditions to hold:

IC1′. All loyal lieutenants attack within 10 minutes of one another.
IC2′. If the commanding general is loyal, then every loyal lieutenant attacks within 10 minutes of the time given in the commander's order.

(We assume that the orders are given and processed the day before the attack and that the time at which an order is received is irrelevant—only the attack time given in the order matters.)

Like the Byzantine Generals Problem, this problem is unsolvable unless more than two-thirds of the generals are loyal. We prove this by first showing that if there were a solution for three generals that coped with one traitor, then we could construct a three-general solution to the Byzantine Generals Problem that also worked in the presence of one traitor. Suppose the commander wishes to send an "attack" or "retreat" order. He orders an attack by sending an attack time of 1:00 and orders a retreat by sending an attack time of 2:00, using the assumed algorithm. Each lieutenant uses the following procedure to obtain his order.

(1) After receiving the attack time from the commander, a lieutenant does one of the following:

    (a) If the time is 1:10 or earlier, then attack.
    (b) If the time is 1:50 or later, then retreat.
    (c) Otherwise, continue to step (2).

(2) Ask the other lieutenant what decision he reached in step (1).

(a) If the other lieutenant reached a decision, then make the same decision he did.
(b) Otherwise, retreat.

It follows from IC2′ that if the commander is loyal, then a loyal lieutenant will obtain the correct order in step (1), so IC2 is satisfied. If the commander is loyal, then IC1 follows from IC2, so we need only prove IC1 under the assumption that the commander is a traitor. Since there is at most one traitor, this means that both lieutenants are loyal. It follows from IC1′ that if one lieutenant decides to attack in step (1), then the other cannot decide to retreat in step (1). Hence, either they will both come to the same decision in step (1) or at least one of them will defer his decision until step (2). In this case, it is easy to see that they both arrive at the same decision, so IC1 is satisfied. We have therefore constructed a three-general solution to the Byzantine Generals Problem that handles one traitor, which is impossible. Hence, we cannot have a three-general algorithm that maintains IC1′ and IC2′ in the presence of a traitor.

The method of having one general simulate m others can now be used to prove that no solution with fewer than $3m + 1$ generals can cope with m traitors. The proof is similar to the one for the original Byzantine Generals Problem and is left to the reader.

3. A SOLUTION WITH ORAL MESSAGES

We showed above that for a solution to the Byzantine Generals Problem using oral messages to cope with m traitors, there must be at least $3m + 1$ generals. We now give a solution that works for $3m + 1$ or more generals. However, we first specify exactly what we mean by "oral messages". Each general is supposed to execute some algorithm that involves sending messages to the other generals, and we assume that a loyal general correctly executes his algorithm. The definition of an oral message is embodied in the following assumptions which we make for the generals' message system:

A1. Every message that is sent is delivered correctly.
A2. The receiver of a message knows who sent it.
A3. The absence of a message can be detected.

Assumptions A1 and A2 prevent a traitor from interfering with the communication between two other generals, since by A1 he cannot interfere with the messages they do send, and by A2 he cannot confuse their intercourse by introducing spurious messages. Assumption A3 will foil a traitor who tries to prevent a decision by simply not sending messages. The practical implementation of these assumptions is discussed in Section 6.

The algorithms in this section and in the following one require that each general be able to send messages directly to every other general. In Section 5, we describe algorithms which do not have this requirement.

A traitorous commander may decide not to send any order. Since the lieutenants must obey some order, they need some default order to obey in this case. We let RETREAT be this default order.

We inductively define the *O*ral *M*essage algorithms OM(m), for all nonnegative integers m, by which a commander sends an order to $n - 1$ lieutenants. We show that OM(m) solves the Byzantine Generals Problem for $3m + 1$ or more generals in the presence of at most m traitors. We find it more convenient to describe this algorithm in terms of the lieutenants "obtaining a value" rather than "obeying an order".

The algorithm assumes a function *majority* with the property that if a majority of the values v_i equal v, then *majority*$(v_1, \ldots, v_{n-1})$ equals v. (Actually, it assumes a sequence of such functions—one for each n.) There are two natural choices for the value of *majority*$(v_1, \ldots, v_{n-1})$:

1. The majority value among the v_i if it exists, otherwise the value RETREAT;
2. The median of the v_i, assuming that they come from an ordered set.

The following algorithm requires only the aforementioned property of *majority*.

Algorithm OM(0).

(1) The commander sends his value to every lieutenant.
(2) Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm OM(m), $m > 0$.

(1) The commander sends his value to every lieutenant.
(2) For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM($m - 1$) to send the value v_i to each of the $n - 2$ other lieutenants.
(3) For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM($m - 1$)), or else RETREAT if he received no such value. Lieutenant i uses the value *majority*$(v_1, \ldots, v_{n-1})$.

To understand how this algorithm works, we consider the case $m = 1$, $n = 4$. Figure 3 illustrates the messages received by Lieutenant 2 when the commander sends the value v and Lieutenant 3 is a traitor. In the first step of OM(1), the commander sends v to all three lieutenants. In the second step, Lieutenant 1 sends the value v to Lieutenant 2, using the trivial algorithm OM(0). Also in the second step, the traitorous Lieutenant 3 sends Lieutenant 2 some other value x. In step 3, Lieutenant 2 then has $v_1 = v_2 = v$ and $v_3 = x$, so he obtains the correct value $v =$ *majority*(v, v, x).

Next, we see what happens if the commander is a traitor. Figure 4 shows the values received by the lieutenants if a traitorous commander sends three arbitrary values x, y, and z to the three lieutenants. Each lieutenant obtains $v_1 = x$, $v_2 = y$, and $v_3 = z$, so they all obtain the same value *majority*(x, y, z) in step (3), regardless of whether or not any of the three values x, y, and z are equal.

The recursive algorithm OM(m) invokes $n - 1$ separate executions of the algorithm OM($m - 1$), each of which invokes $n - 2$ executions of OM($m - 2$), etc. This means that, for $m > 1$, a lieutenant sends many separate messages to each other lieutenant. There must be some way to distinguish among these different messages. The reader can verify that all ambiguity is removed if each lieutenant i prefixes the number i to the value v_i that he sends in step (2). As the recursion "unfolds," the algorithm OM($m - k$) will be called $(n - 1) \cdots (n - k)$ times to send a value prefixed by a sequence of k lieutenants' numbers.

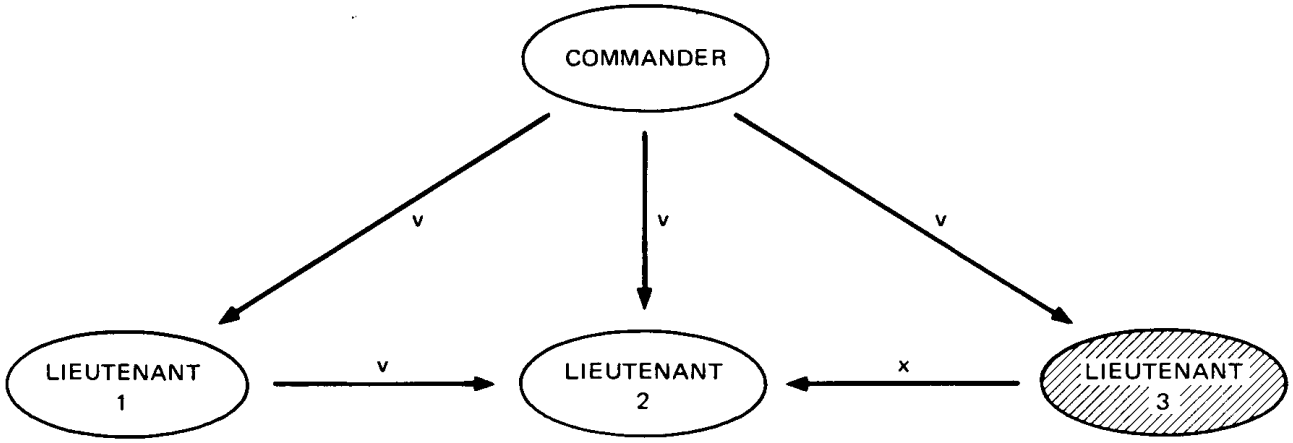


Fig. 3. Algorithm OM(1); Lieutenant 3 a traitor.

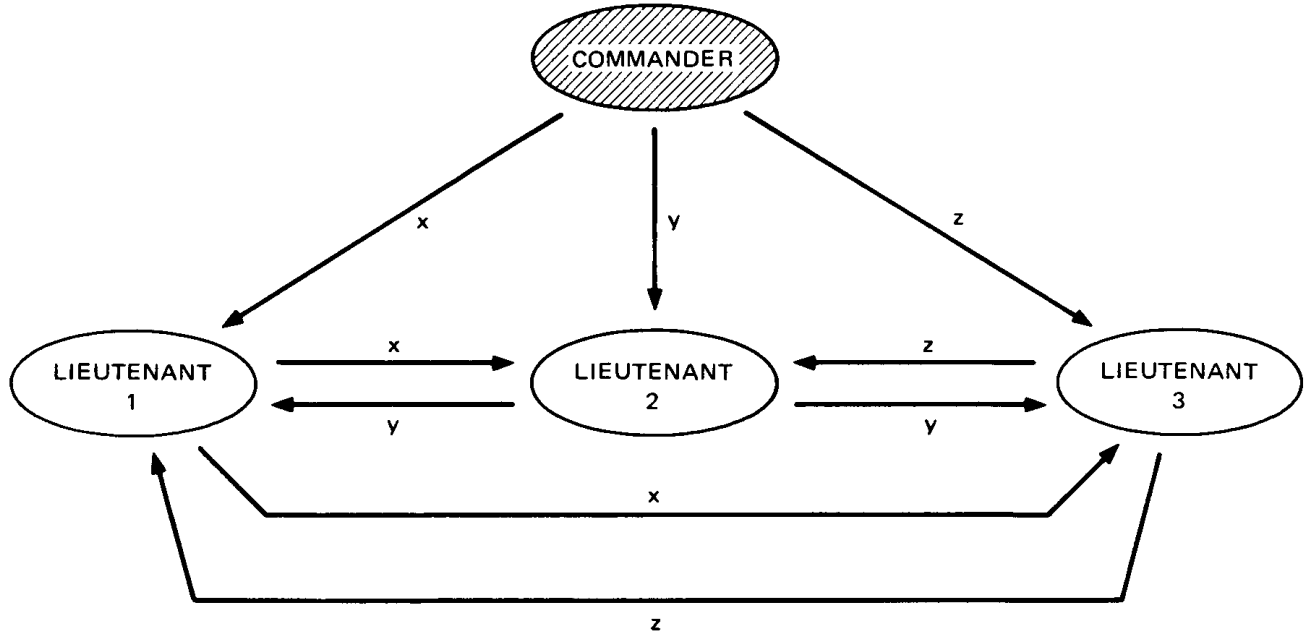


Fig. 4. Algorithm OM(1); the commander a traitor.

To prove the correctness of the algorithm OM(m) for arbitrary m, we first prove the following lemma.

LEMMA 1. *For any m and k, Algorithm OM(m) satisfies IC2 if there are more than $2k + m$ generals and at most k traitors.*

PROOF. The proof is by induction on m. IC2 only specifies what must happen if the commander is loyal. Using A1, it is easy to see that the trivial algorithm OM(0) works if the commander is loyal, so the lemma is true for $m = 0$. We now assume it is true for $m - 1$, $m > 0$, and prove it for m.

In step (1), the loyal commander sends a value v to all $n - 1$ lieutenants. In step (2), each loyal lieutenant applies OM($m - 1$) with $n - 1$ generals. Since by hypothesis $n > 2k + m$, we have $n - 1 > 2k + (m - 1)$, so we can apply the induction hypothesis to conclude that every loyal lieutenant gets $v_j = v$ for each loyal Lieutenant j. Since there are at most k traitors, and $n - 1 > 2k + (m - 1) \geq 2k$, a majority of the $n - 1$ lieutenants are loyal. Hence, each loyal lieutenant has $v_i = v$ for a majority of the $n - 1$ values i, so he obtains *majority*$(v_1, \ldots, v_{n-1}) = v$ in step (3), proving IC2. □

The following theorem asserts that Algorithm OM(m) solves the Byzantine Generals Problem.

THEOREM 1. *For any m, Algorithm OM(m) satisfies conditions IC1 and IC2 if there are more than $3m$ generals and at most m traitors.*

PROOF. The proof is by induction on m. If there are no traitors, then it is easy to see that OM(0) satisfies IC1 and IC2. We therefore assume that the theorem is true for OM($m - 1$) and prove it for OM(m), $m > 0$.

We first consider the case in which the commander is loyal. By taking k equal to m in Lemma 1, we see that OM(m) satisfies IC2. IC1 follows from IC2 if the commander is loyal, so we need only verify IC1 in the case that the commander is a traitor.

There are at most m traitors, and the commander is one of them, so at most $m - 1$ of the lieutenants are traitors. Since there are more than $3m$ generals, there are more than $3m - 1$ lieutenants, and $3m - 1 > 3(m - 1)$. We may therefore apply the induction hypothesis to conclude that OM($m - 1$) satisfies conditions IC1 and IC2. Hence, for each j, any two loyal lieutenants get the same value for v_j in step (3). (This follows from IC2 if one of the two lieutenants is Lieutenant j, and from IC1 otherwise.) Hence, any two loyal lieutenants get the same vector of values $v_1, \ldots, v_{n-1}$, and therefore obtain the same value *majority*$(v_1, \ldots, v_{n-1})$ in step (3), proving IC1. □

4. A SOLUTION WITH SIGNED MESSAGES

As we saw from the scenario of Figures 1 and 2, it is the traitors' ability to lie that makes the Byzantine Generals Problem so difficult. The problem becomes easier to solve if we can restrict that ability. One way to do this is to allow the generals to send unforgeable signed messages. More precisely, we add to A1–A3 the

following assumption:

A4 (a) A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
(b) Anyone can verify the authenticity of a general's signature.

Note that we make no assumptions about a traitorous general's signature. In particular, we allow his signature to be forged by another traitor, thereby permitting collusion among the traitors.

Now that we have introduced signed messages, our previous argument that four generals are required to cope with one traitor no longer holds. In fact, a three-general solution does exist. We now give an algorithm that copes with m traitors for any number of generals. (The problem is vacuous if there are fewer than $m + 2$ generals.)

In our algorithm, the commander sends a signed order to each of his lieutenants. Each lieutenant then adds his signature to that order and sends it to the other lieutenants, who add their signatures and send it to others, and so on. This means that a lieutenant must effectively receive one signed message, make several copies of it, and sign and send those copies. It does not matter how these copies are obtained; a single message might be photocopied, or else each message might consist of a stack of identical messages which are signed and distributed as required.

Our algorithm assumes a function *choice* which is applied to a set of orders to obtain a single one. The only requirements we make for this function are

1. If the set V consists of the single element v, then $choice(V) = v$.
2. $choice(\varnothing) =$ RETREAT, where $\varnothing$ is the empty set.

Note that one possible definition is to let $choice(V)$ be the median element of V—assuming that there is an ordering of the elements.

In the following algorithm, we let $x:i$ denote the value x signed by General i. Thus, $v:j:i$ denotes the value v signed by j, and then that value $v:j$ signed by i. We let General 0 be the commander. In this algorithm, each lieutenant i maintains a set V_i, containng the set of properly signed orders he has received so far. (If the commander is loyal, then this set should never contain more than a single element.) Do not confuse V_i, the set of *orders* he has received, with the set of messages that he has received. There may be many different messages with the same order.

Algorithm SM(m).

Initially $V_i = \varnothing$.

(1) The commander signs and sends his value to every lieutenant.
(2) For each i:

(A) If Lieutenant i receives a message of the form $v:0$ from the commander and he has not yet received any order, then

(i) he lets V_i equal $\{v\}$;
(ii) he sends the message $v:0:i$ to every other lieutenant.

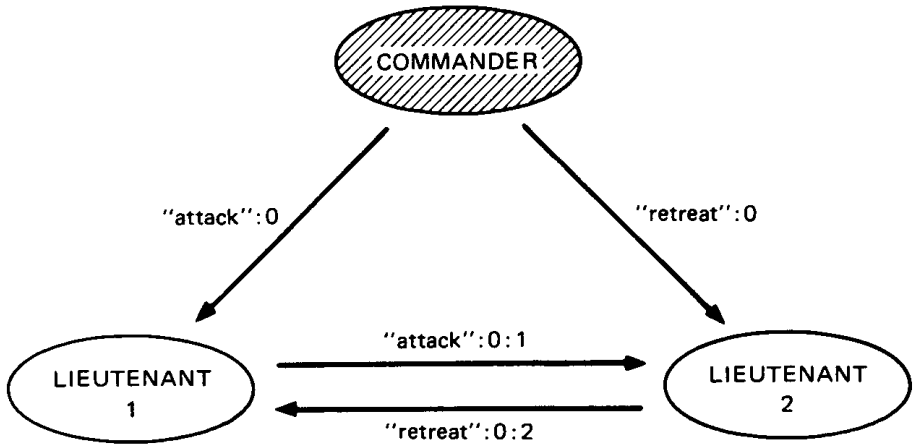


Fig. 5. Algorithm SM(1); the commander a traitor.

(B) If Lieutenant i receives a message of the form $v:0:j_1:\cdots:j_k$ and v is not in the set V_i, then

(i) he adds v to V_i;

(ii) if $k < m$, then he sends the message $v:0:j_1:\cdots:j_k:i$ to every lieutenant other than $j_1, \ldots, j_k$.

(3) For each i: When Lieutenant i will receive no more messages, he obeys the order $choice(V_i)$.

Note that in step (2), Lieutenant i ignores any message containing an order v that is already in the set V_i.

We have not specified how a lieutenant determines in step (3) that he will receive no more messages. By induction on k, one easily shows that for each sequence of lieutenants $j_1, \ldots, j_k$ with $k \leq m$, a lieutenant can receive at most one message of the form $v:0:j_1:\cdots:j_k$ in step (2). If we require that Lieutenant j_k either send such a message or else send a message reporting that he will not send such a message, then it is easy to decide when all messages have been received. (By assumption A3, a lieutenant can determine if a traitorous lieutenant j_k sends neither of those two messages.) Alternatively, time-out can be used to determine when no more messages will arrive. The use of time-out is discussed in Section 6.

Note that in step (2), Lieutenant i ignores any messages that do not have the proper form of a value followed by a string of signatures. If packets of identical messages are used to avoid having to copy messages, this means that he throws away any packet that does not consist of a sufficient number of identical, properly signed messages. (There should be $(n-k-2)(n-k-3)\cdots(n-m-2)$ copies of the message if it has been signed by k lieutenants.)

Figure 5 illustrates Algorithm SM(1) for the case of three generals when the commander is a traitor. The commander sends an "attack" order to one lieutenant and a "retreat" order to the other. Both lieutenants receive the two orders in step (2), so after step (2) $V_1 = V_2 = \{$"attack", "retreat"$\}$, and they both obey the order $choice(\{$"attack", "retreat"$\})$. Observe that here, unlike the situation in Figure 2, the lieutenants know the commander is a traitor because his signature

appears on two different orders, and A4 states that only he could have generated those signatures.

In Algorithm SM(m), a lieutenant signs his name to acknowledge his receipt of an order. If he is the mth lieutenant to add his signature to the order, then that signature is not relayed to anyone else by its recipient, so it is superfluous. (More precisely, assumption A2 makes it unnecessary.) In particular, the lieutenants need not sign their messages in SM(1).

We now prove the correctness of our algorithm.

THEOREM 2. *For any m, Algorithm SM(m) solves the Byzantine Generals Problem if there are at most m traitors.*

PROOF. We first prove IC2. If the commander is loyal, then he sends his signed order $v:0$ to every lieutenant in step (1). Every loyal lieutenant will therefore receive the order v in step (2)(A). Moreover, since no traitorous lieutenant can forge any other message of the form $v':0$, a loyal lieutenant can receive no additional order in step (2)(B). Hence, for each loyal Lieutenant i, the set V_i obtained in step (2) consists of the single order v, which he will obey in step (3) by property 1 of the *choice* function. This proves IC2.

Since IC1 follows from IC2 if the commander is loyal, to prove IC1 we need only consider the case in which the commander is a traitor. Two loyal lieutenants i and j obey the same order in step (3) if the sets of orders V_i and V_j that they receive in step (2) are the same. Therefore, to prove IC1 it suffices to prove that, if i puts an order v into V_i in step (2), then j must put the same order v into V_j in step (2). To do this, we must show that j receives a properly signed message containing that order. If i receives the order v in step (2)(A), then he sends it to j in step (2)(A)(ii); so j receives it (by A1). If i adds the order to V_i in step (2)(B), then he must receive a first message of the form $v:0:j_1:\cdots:j_k$. If j is one of the j_r, then by A4 he must already have received the order v. If not, we consider two cases:

1. $k < m$. In this case, i sends the message $v:0:j_1:\cdots:j_k:i$ to j; so j must receive the order v.
2. $k = m$. Since the commander is a traitor, at most $m - 1$ of the lieutenants are traitors. Hence, at least one of the lieutenants $j_1, \ldots, j_m$ is loyal. This loyal lieutenant must have sent j the value v when he first received it, so j must therefore receive that value.

This completes the proof. □

5. MISSING COMMUNICATION PATHS

Thus far, we have assumed that a general can send messages directly to every other general. We now remove this assumption. Instead, we suppose that physical barriers place some restrictions on who can send messages to whom. We consider the generals to form the nodes of a simple,[2] finite undirected graph G, where an arc between two nodes indicates that those two generals can send messages

[2] A simple graph is one in which there is at most one arc joining any two nodes, and every arc connects two distinct nodes.

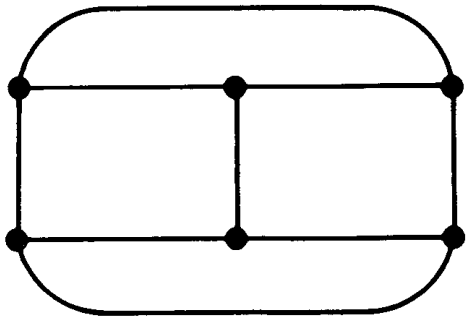


Fig. 6. A 3-regular graph.

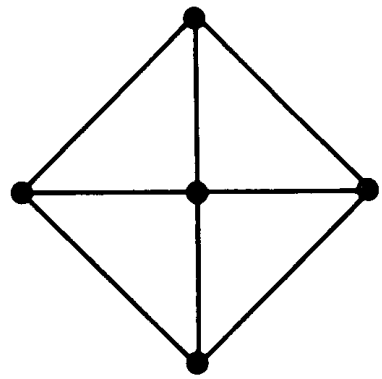


Fig. 7. A graph that is not 3-regular.

directly to one another. We now extend Algorithms OM(m) and SM(m), which assumed G to be completely connected, to more general graphs.

To extend our oral message algorithm OM(m), we need the following definition, where two generals are said to be *neighbors* if they are joined by an arc.

Definition 1.

(a) A set of nodes $\{i_1, \ldots, i_p\}$ is said to be a *regular set of neighbors* of a node i if

(i) each i_j is a neighbor of i, and
(ii) for any general k different from i, there exist paths $\gamma_{j,k}$ from i_j to k not passing through i such that any two different paths $\gamma_{j,k}$ have no node in common other than k.

(b) The graph G is said to be *p-regular* if every node has a regular set of neighbors consisting of p distinct nodes.

Figure 6 shows an example of a simple 3-regular graph. Figure 7 shows an example of a graph that is not 3-regular because the central node has no regular set of neighbors containing three nodes.

We extend OM(m) to an algorithm that solves the Byzantine Generals Problem in the presence of m traitors if the graph G of generals is $3m$-regular. (Note that a $3m$-regular graph must contain at least $3m + 1$ nodes.) For all *positive* integers m and p, we define the algorithm OM(m, p) as follows when the graph G of generals is p-regular. (OM(m, p) is not defined if G is not p-regular.) The definition uses induction on m.

Algorithm *OM*(m, p).

(0) Choose a regular set N of neighbors of the commander consisting of p lieutenants.
(1) The commander sends his value to every lieutenant in N.
(2) For each i in N, let v_i be the value Lieutenant i receives from the commander, or else RETREAT if he receives no value. Lieutenant i sends v_i to every other lieutenant k as follows:

  (A) If $m = 1$, then by sending the value along the path $\gamma_{i,k}$ whose existence is guaranteed by part (a)(ii) of Definition 1.
  (B) If $m > 1$, then by acting as the commander in the algorithm OM($m - 1, p - 1$), with the graph of generals obtained by removing the original commander from G.

(3) For each k, and each i in N with $i \neq k$, let v_i be the value Lieutenant k received from Lieutenant i in step (2), or RETREAT if he received no value. Lieutenant k uses the value $majority(v_{i_1}, \ldots, v_{i_p})$, where $N = \{i_1, \ldots, i_p\}$.

Note that removing a single node from a p-regular graph leaves a $(p - 1)$-regular graph. Hence, one can apply the algorithm OM$(m - 1, p - 1)$ in step (2)(B).

We now prove that OM$(m, 3m)$ solves the Byzantine Generals Problem if there are at most m traitors. The proof is similar to the proof for the algorithm OM(m) and will just be sketched. It begins with the following extension of Lemma 1.

LEMMA 2. *For any $m > 0$ and any $p \geq 2k + m$, Algorithm OM(m, p) satisfies IC2 if there are at most k traitors.*

PROOF. For $m = 1$, observe that a lieutenant obtains the value $majority(v_1, \ldots, v_p)$, where each v_i is a value sent to him by the commander along a path disjoint from the path used to send the other values to him. Since there are at most k traitors and $p \geq 2k + 1$, more than half of those paths are composed entirely of loyal lieutenants. Hence, if the commander is loyal, then a majority of the values v_i will equal the value he sent, which implies that IC2 is satisfied.

Now assume the lemma for $m - 1$, $m > 1$. If the commander is loyal, then each of the p lieutenants in N gets the correct value. Since $p > 2k$, a majority of them are loyal, and by the induction hypothesis each of them sends the correct value to every loyal lieutenant. Hence, each loyal lieutenant gets a majority of correct values, thereby obtaining the correct value in step (3). □

The correctness of Algorithm OM$(m, 3m)$ is an immediate consequence of the following result.

THEOREM 3. *For any $m > 0$ and any $p \geq 3m$, Algorithm OM(m, p) solves the Byzantine Generals Problem if there are at most m traitors.*

PROOF. By Lemma 2, letting $k = m$, we see that OM(m, p) satisfies IC2. If the commander is loyal, then IC1 follows from IC2, so we need only prove IC1 under the assumption that the commander is a traitor. To do this, we prove that every loyal lieutenant gets the same set of values v_i in step (3). If $m = 1$, then this follows because all the lieutenants, including those in N, are loyal and the paths $\gamma_{i,k}$ do not pass through the commander. For $m > 1$, a simple induction argument can be applied, since $p \geq 3m$ implies that $p - 1 \geq 3(m - 1)$. □

Our extension of Algorithm OM(m) requires that the graph G be $3m$-regular, which is a rather strong connectivity hypothesis.[3] In fact, if there are only $3m + 1$ generals (the minimum number required), then $3m$-regularity means complete connectivity, and Algorithm OM$(m, 3m)$ reduces to Algorithm OM(m). In contrast, Algorithm SM(m) is easily extended to allow the weakest possible connectivity hypothesis. Let us first consider how much connectivity is needed for the Byzantine Generals Problem to be solvable. IC2 requires that a loyal lieutenant obey a loyal commander. This is clearly impossible if the commander cannot communicate with the lieutenant. In particular, if every message from the

[3] A recent algorithm of Dolev [2] requires less connectivity.

commander to the lieutenant must be relayed by traitors, then there is no way to guarantee that the lieutenant gets the commander's order. Similarly, IC1 cannot be guaranteed if there are two lieutenants who can only communicate with one another via traitorous intermediaries.

The weakest connectivity hypothesis for which the Byzantine Generals Problem is solvable is that the subgraph formed by the loyal generals be connected. We show that under this hypothesis, the algorithm SM($n - 2$) is a solution, where n is the number of generals—regardless of the number of traitors. Of course, we must modify the algorithm so that generals only send messages to where they can be sent. More precisely, in step (1), the commander sends his signed order only to his neighboring lieutenants; and in step (2)(B), Lieutenant i only sends the message to every *neighboring* lieutenant not among the j_r.

We prove the following more general result, where the *diameter* of a graph is the smallest number d such that any two nodes are connected by a path containing at most d arcs.

THEOREM 4. *For any m and d, if there are at most m traitors and the subgraph of loyal generals has diameter d, then Algorithm SM($m + d - 1$) (with the above modification) solves the Byzantine Generals Problem.*

PROOF. The proof is quite similar to that of Theorem 2 and is just sketched here. To prove IC2, observe that by hypothesis there is a path from the loyal commander to a lieutenant i going through $d - 1$ or fewer loyal lieutenants. Those lieutenants will correctly relay the order until it reaches i. As before, assumption A4 prevents a traitor from forging a different order.

To prove IC1, we assume the commander is a traitor and must show that any order received by a loyal lieutenant i is also received by a loyal lieutenant j. Suppose i receives an order $v:0:j_1: \cdots :j_k$ not signed by j. If $k < m$, then i will send it to every neighbor who has not already received that order, and it will be relayed to j within $d - 1$ more steps. If $k \geq m$, then one of the first m signers must be loyal and must have sent it to all of his neighbors, whereupon it will be relayed by loyal generals and will reach j within $d - 1$ steps. □

COROLLARY. *If the graph of loyal generals is connected, then SM($n - 2$) (as modified above) solves the Byzantine Generals Problem for n generals.*

PROOF. Let d be the diameter of the graph of loyal generals. Since the diameter of a connected graph is less than the number of nodes, there must be more than d loyal generals and fewer than $n - d$ traitors. The result follows from the theorem by letting $m = n - d - 1$. □

Theorem 4 assumes that the subgraph of loyal generals is connected. Its proof is easily extended to show that even if this is not the case, if there are at most m traitors, then the algorithm SM($m + d - 1$) has the following two properties:

1. Any two loyal generals connected by a path of length at most d passing through only loyal generals will obey the same order.
2. If the commander is loyal, then any loyal lieutenant connected to him by a path of length at most $m + d$ passing only through loyal generals will obey his order.

6. RELIABLE SYSTEMS

Other than using intrinsically reliable circuit components, the only way we know to implement a reliable computer system is to use several different "processors" to compute the same result, and then to perform a majority vote on their outputs to obtain a single value. (The voting may be performed within the system, or externally by the users of the output.) This is true whether one is implementing a reliable computer using redundant circuitry to protect against the failure of individual chips, or a ballistic missile defense system using redundant computing sites to protect against the destruction of individual sites by a nuclear attack. The only difference is in the size of the replicated "processor".

The use of majority voting to achieve reliability is based upon the assumption that all the nonfaulty processors will produce the same output. This is true so long as they all use the same input. However, any single input datum comes from a single physical component—for example, from some other circuit in the reliable computer, or from some radar site in the missile defense system—and a malfunctioning component can give different values to different processors. Moreover, different processors can get different values even from a nonfaulty input unit if they read the value while it is changing. For example, if two processors read a clock while it is advancing, then one may get the old time and the other the new time. This can only be prevented by synchronizing the reads with the advancing of the clock.

In order for majority voting to yield a reliable system, the following two conditions should be satisfied:

1. All nonfaulty processors must use the same input value (so they produce the same output).
2. If the input unit is nonfaulty, then all nonfaulty processes use the value it provides as input (so they produce the correct output).

These are just our interactive consistency conditions IC1 and IC2, where the "commander" is the unit generating the input, the "lieutenants" are the processors, and "loyal" means nonfaulty.

It is tempting to try to circumvent the problem with a "hardware" solution. For example, one might try to insure that all processors obtain the same input value by having them all read it from the same wire. However, a faulty input unit could send a marginal signal along the wire—a signal that can be interpreted by some processors as a 0 and by others as a 1. There is no way to guarantee that different processors will get the same value from a possibly faulty input device except by having the processors communicate among themselves to solve the Byzantine Generals Problem.

Of course, a faulty input device may provide meaningless input values. All that a Byzantine Generals solution can do is guarantee that all processors use the same input value. If the input is an important one, then there should be several separate input devices providing redundant values. For example, there should be redundant radars as well as redundant processing sites in a missile defense system. However, redundant inputs cannot achieve reliability; it is still necessary to insure that the nonfaulty processors use the redundant data to produce the same output.

In case the input device is nonfaulty but gives different values because it is read while its value is changing, we still want the nonfaulty processors to obtain a reasonable input value. It can be shown that, if the functions *majority* and *choice* are taken to be the median functions, then our algorithms have the property that the value obtained by the nonfaulty processors lies within the range of values provided by the input unit. Thus, the nonfaulty processors will obtain a reasonable value so long as the input unit produces a reasonable range of values.

We have given several solutions, but they have been stated in terms of Byzantine generals rather than in terms of computing systems. We now examine how these solutions can be applied to reliable computing systems. Of course, there is no problem implementing a "general's" algorithm with a processor. The problems lie in implementing a message passing system that meets assumptions A1–A3 (assumptions A1–A4 for Algorithm SM(m)). We now consider these assumptions in order.

A1. Assumption A1 states that every message sent by a nonfaulty processor is delivered correctly. In real systems, communication lines can fail. For the oral message algorithms OM(m) and OM(m, p), the failure of the communication line joining two processors is indistinguishable from the failure of one of the processors. Hence, we can only guarantee that these algorithms will work in the presence of up to m failures, be they processor or communication line failures. (Of course, the failure of several communication lines attached to the same processor is equivalent to a single processor failure.) If we assume that a failed communication line cannot result in the forgery of a signed message—an assumption which we will see below is quite reasonable, then our signed message algorithm SM(m) is insensitive to communication line failure. More precisely, Theorem 4 remains valid even with communication line failure. A failed communication line has the same effect as simply removing the communication line—it lowers the connectivity of the processor graph.

A2. Assumption A2 states that a processor can determine the originator of any message that it received. What is actually necessary is that a faulty processor not be able to impersonate a nonfaulty one. In practice, this means that interprocess communication be over fixed lines rather than through some message switching network. (If a switching network is used, then faulty network nodes must be considered, and the Byzantine Generals Problem appears again.) Note that assumption A2 is not needed if A4 is assumed and all messages are signed, since impersonation of another processor would imply forging its messages.

A3. Assumption A3 requires that the absence of a message can be detected. The absence of a message can only be detected by its failure to arrive within some fixed length of time—in other words, by the use of some time-out convention. The use of time-out to satisfy A3 requires two assumptions:

1. There is a fixed maximum time needed for the generation and transmission of a message.
2. The sender and receiver have clocks that are synchronized to within some fixed maximum error.

The need for the first assumption is fairly obvious, since the receiver must know

how long he needs to wait for the message to arrive. (The generation time is how long it takes the processor to send the message after receiving all the input necessary to generate it.) The need for the second assumption is less obvious. However, it can be shown that either this assumption or an equivalent one is necessary to solve the Byzantine Generals Problem. More precisely, suppose that we allow algorithms in which the generals take action only in the following circumstances:

1. At some fixed initial time (the same for all generals).
2. Upon the receipt of a message.
3. When a randomly chosen length of time has elapsed. (I.e., a general can set a timer to a random value and act when the timer goes off.)

(This yields the most general class of algorithms we can envision which does not allow the construction of synchronized clocks.) It can be shown that no such algorithm can solve the Byzantine Generals Problem if messages can be transmitted arbitrarily quickly, even if there is an upper bound on message transmission delay. Moreover, no solution is possible even if we restrict the traitors so that the only incorrect behavior they are permitted is the failure to send a message. The proof of this result is beyond the scope of this paper. Note that placing a lower as well as an upper bound on transmission delay allows processors to implement clocks by sending messages back and forth.

The above two assumptions make it easy to detect unsent messages. Let μ be the maximum message generation and transmission delay, and assume the nonfaulty processors have clocks that differ from one another by at most τ at any time. Then any message that a nonfaulty process should begin to generate by time T on its clock will arrive at its destination by time $T + \mu + \tau$ on the receiver's clock. Hence, if the receiver has not received the message by that time, then it may assume that it was not sent. (If it arrives later, then the sender must be faulty, so the correctness of our algorithms does not depend upon the message being sent.) By fixing the time at which the input processor sends its value, one can calculate until what time on its own clock a processor must wait for each message. For example, in Algorithm SM(m) a processor must wait until time $T_0 + k(\mu + \tau)$ for any message having k signatures, where T_0 is the time (on his clock) at which the commander starts executing the algorithm.

No two clocks run at precisely the same rate, so no matter how accurately the processors' clocks are synchronized initially, they will eventually drift arbitrarily far apart unless they are periodically resynchronized. We therefore have the problem of keeping the processors' clocks all synchronized to within some fixed amount, even if some of the processors are faulty. This is as difficult a problem as the Byzantine Generals Problem itself. Solutions to the clock synchronization problem exist which are closely related to our Byzantine Generals solutions. They will be described in a future paper.

A4. Assumption A4 requires that processors be able to sign their messages in such a way that a nonfaulty processor's signature cannot be forged. A signature is a piece of redundant information $S_i(M)$ generated by process i from a data item M. A message signed by i consists of a pair $(M, S_i(M))$. To meet parts (a)

and (b) of A4, the function S_i must have the following two properties:

(a) If processor i is nonfaulty, then no faulty processor can generate $S_i(M)$.
(b) Given M and X, any process can determine if X equals $S_i(M)$.

Property (a) can never be guaranteed, since $S_i(M)$ is just a data item, and a faulty processor could generate any data item. However, we can make the probability of its violation as small as we wish, thereby making the system as reliable as we wish. How this is done depends upon the type of faults we expect to encounter. There are two cases of interest:

1. *Random Malfunction.* By making S_i a suitably "randomizing" function, we can make the probability that a random malfunction in a processor generates a correct signature essentially equal to the probability of its doing so through a random choice procedure—that is, the reciprocal of the number of possible signatures. The following is one method for doing this. Assume that messages are encoded as positive integers less than P, where P is a power of two. Let $S_i(M)$ equal $M * K_i \bmod P$, where K_i is a randomly chosen odd number less than P. Letting K_i^{-1} be the unique number less than P such that $K_i * K_i^{-1} \equiv 1 \bmod P$, a process can check that $X = S_i(M)$ by testing that $M \equiv X * K_i^{-1} \bmod P$. If another processor does not have K_i in its memory, then the probability of its generating the correct signature $M * K_i$ for a single (nonzero) message M should be $1/P$: its probability of doing so by random choice. (Note that if the processor could obtain K_i by some simple procedure, then there might be a larger probability of a faulty processor j forging i's signature by substituting K_i for K_j when trying to compute $S_j(M)$.)

2. *Malicious Intelligence.* If the faulty processor is being guided by a malicious intelligence—for example, if it is a perfectly good processor being operated by a human who is trying to disrupt the system—then the construction of the signature function S_i becomes a cryptography problem. We refer the reader to [1] and [4] for a discussion of how this problem can be solved.

Note that it is easy to generate the signature $S_i(M)$ if the process has already seen that signature. Hence, it is important that the same message never have to be signed twice. This means that, when using SM(m) repeatedly to distribute a sequence of values, sequence numbers should be appended to the values to guarantee uniqueness.

7. CONCLUSION

We have presented several solutions to the Byzantine Generals Problem, under various hypotheses, and shown how they can be used in implementing reliable computer systems. These solutions are expensive in both the amount of time and the number of messages required. Algorithms OM(m) and SM(m) both require message paths of length up to $m + 1$. In other words, each lieutenant may have to wait for messages that originated at the commander and were then relayed via m other lieutenants. Fischer and Lynch have shown that this must be true for any solution that can cope with m traitors, so our solutions are optimal in that respect. Our algorithms for a graph that is not completely connected require

message paths of length up to $m + d$, where d is the diameter of the subgraph of loyal generals. We suspect that this is also optimal.

Algorithms OM(m) and SM(m) involve sending up to $(n - 1)(n - 2) \cdots (n - m - 1)$ messages. The number of separate messages required can certainly be reduced by combining messages. It may also be possible to reduce the amount of information transferred, but this has not been studied in detail. However, we expect that a large number of messages will still be required.

Achieving reliability in the face of arbitrary malfunctioning is a difficult problem, and its solution seems to be inherently expensive. The only way to reduce the cost is to make assumptions about the type of failure that may occur. For example, it is often assumed that a computer may fail to respond but will never respond incorrectly. However, when extremely high reliability is required, such assumptions cannot be made, and the full expense of a Byzantine Generals solution is required.

REFERENCES

1. Diffie, W., and Hellman, M.E. New directions in cryptography. *IEEE Trans. Inf. Theory IT-22* (Nov. 1976), 644–654.
2. Dolev, D. The Byzantine generals strike again. *J. Algorithms 3*, 1 (Jan. 1982).
3. Pease, M., Shostak, R., and Lamport, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.
4. Rivest, R.L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb. 1978), 120–126.

Received April 1980; revised November 1981; accepted November 1981

Chapter 4

The Part-Time Parliament

Authors:
Leslie Lamport

Publication year: 1998

The Part-Time Parliament

Leslie Lamport

This article appeared in *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133-169. Minor corrections were made on 29 August 2000.

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lampson [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

Keith Marzullo
University of California, San Diego

Authors' address: Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1998 ACM 0000-0000/98/0000-0000 $00.00

1 The Problem

1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.[1] Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing. The Paxons' solution may therefore be of some interest to computer scientists. I present here a short history of the Paxos Parliament's protocol, followed by an even shorter discussion of its relevance for distributed systems.

Paxon civilization was destroyed by a foreign invasion, and archeologists have just recently begun to unearth its history. Our knowledge of the Paxon Parliament is therefore fragmentary. Although the basic protocols are known, we are ignorant of many details. Where such details are of interest, I will take the liberty of speculating on what the Paxons might have done.

1.2 Requirements

Parliament's primary task was to determine the law of the land, which was defined by the sequence of decrees it passed. A modern parliament will employ a secretary to record its actions, but no one in Paxos was willing to remain in the Chamber throughout the session to act as secretary. Instead, each Paxon legislator maintained a *ledger* in which he recorded the numbered sequence of decrees that were passed. For example, legislator $\Lambda\check{\imath}\nu\chi\partial$'s ledger had the entry

155: *The olive tax is 3 drachmas per ton*

if she believed that the 155th decree passed by Parliament set the tax on olives to 3 drachmas per ton. Ledgers were written with indelible ink, and their entries could not be changed.

The first requirement of the parliamentary protocol was the *consistency of ledgers*, meaning that no two ledgers could contain contradictory information. If legislator $\Phi\iota\sigma\partial\epsilon\rho$ had the entry

132: *Lamps must use only olive oil*

in his ledger, then no other legislator's ledger could have a different entry for decree 132. However, another legislator might have no entry in his ledger for decree 132 if he hadn't yet learned that the decree had been passed.

Consistency of ledgers was not sufficient, since it could be trivially fulfilled by leaving all ledgers blank. Some requirement was needed to guarantee that decrees

[1] It should not be confused with the Ionian island of Paxoi, whose name is sometimes corrupted to *Paxos*.

were eventually passed and recorded in ledgers. In modern parliaments, the passing of decrees is hindered by disagreement among legislators. This was not the case in Paxos, where an atmosphere of mutual trust prevailed. Paxon legislators were willing to pass any decree that was proposed. However, their peripatetic propensity posed a problem. Consistency would be lost if one group of legislators passed the decree

37: *Painting on temple walls is forbidden*

and then left for a banquet, whereupon a different group of legislators entered the Chamber and, knowing nothing about what had just happened, passed the conflicting decree

37: *Freedom of artistic expression is guaranteed*

Progress could not be guaranteed unless enough legislators stayed in the Chamber for a long enough time. Because Paxon legislators were unwilling to curtail their outside activities, it was impossible to ensure that any decree would ever be passed. However, legislators were willing to guarantee that, while in the Chamber, they and their aides would act promptly on all parliamentary matters. This guarantee allowed the Paxons to devise a parliamentary protocol satisfying the following *progress condition*.

> If a majority of the legislators[2] were in the Chamber and no one entered or left the Chamber for a sufficiently long period of time then any decree proposed by a legislator in the Chamber would be passed, and every decree that had been passed would appear in the ledger of every legislator in the Chamber.

1.3 Assumptions

The requirements of the parliamentary protocol could be achieved only by providing the legislators with the necessary resources. Each legislator received a sturdy ledger in which to record the decrees, a pen, and a supply of indelible ink. Legislators might forget what they had been doing if they left the Chamber,[3] so they would write notes in the back of the ledgers to remind themselves of important parliamentary tasks. An entry in the list of decrees was never changed, but notes could be crossed out. Achieving the progress condition required that legislators be able to measure the passage of time, so they were given simple hourglass timers.

Legislators carried their ledgers at all times, and could always read the list of decrees and any note that had not been crossed out. The ledgers were made of the finest parchment and were used for only the most important notes. A legislator would write other notes on a slip of paper, which he might (or might not) lose if he left the Chamber.

The acoustics of the Chamber were poor, making oratory impossible. Legislators could communicate only by messenger, and were provided with funds to hire as

[2]In translating the progress condition, I have rendered the Paxon word *μαδζδωριπΐσετ* as *majority of the legislators*. Alternative translations of this word have been proposed and are discussed in Section 2.2.

[3]In one tragic incident, legislator *Τωυεγ* developed irreversible amnesia after being hit on the head by a falling statue just outside the Chamber.

many messengers as they needed. A messenger could be counted on not to garble messages, but he might forget that he had already delivered a message, and deliver it again. Like the legislators they served, messengers devoted only part of their time to parliamentary duties. A messenger might leave the Chamber to conduct some business—perhaps taking a six-month voyage—before delivering a message. He might even leave forever, in which case the message would never be delivered.

Although legislators and messengers could enter and leave at any time, when inside the Chamber they devoted themselves to the business of Parliament. While they remained in the Chamber, messengers delivered messages in a timely fashion and legislators reacted promptly to any messages they received.

The official records of Paxos claim that legislators and messengers were scrupulously honest and strictly obeyed parliamentary protocol. Most scholars discount this as propaganda, intended to portray Paxos as morally superior to its eastern neighbors. Dishonesty, although rare, undoubtedly did occur. However, because it was never mentioned in official documents, we have little knowledge of how Parliament coped with dishonest legislators or messengers. What evidence has been uncovered is discussed in Section 3.3.5.

2 The Single-Decree Synod

The Paxon Parliament evolved from an earlier ceremonial Synod of priests that was convened every 19 years to choose a single, symbolic decree. For centuries, the Synod had chosen the decree by a conventional procedure that required all priests to be present. But as commerce flourished, priests began wandering in and out of the Chamber while the Synod was in progress. Finally, the old protocol failed, and a Synod ended with no decree chosen. To prevent a repetition of this theological disaster, Paxon religious leaders asked mathematicians to formulate a protocol for choosing the Synod's decree. The protocol's requirements and assumptions were essentially the same as those of the later Parliament except that instead of containing a sequence of decrees, a ledger would have at most one decree. The resulting Synod protocol is described here; the Parliamentary protocol is described in Section 3.

Mathematicians derived the Synod protocol in a series of steps. First, they proved results showing that a protocol satisfying certain constraints would guarantee consistency and allow progress. A *preliminary protocol* was then derived directly from these constraints. A restricted version of the preliminary protocol provided the *basic protocol* that guaranteed consistency, but not progress. The complete Synod protocol, satisfying the consistency and progress requirements, was obtained by restricting the basic protocol.[4]

The mathematical results are described in Section 2.1, and the protocols are described informally in Sections 2.2–2.4. A more formal description and correctness proof of the basic protocol appears in the appendix.

[4]The complete history of the Synod protocol's discovery is not known. Like modern computer scientists, Paxon mathematicians would describe elegant, logical derivations that bore no resemblance to how the algorithms were actually derived. However, it is known that the mathematical results (Theorems 1 and 2 of Section 2.1) really did precede the protocol. They were discovered when mathematicians, in response to the request for a protocol, were attempting to prove that a satisfactory protocol was impossible.

2.1 Mathematical Results

The Synod's decree was chosen through a series of numbered *ballots*, where a ballot was a referendum on a single decree. In each ballot, a priest had the choice only of voting for the decree or not voting.[5] Associated with a ballot was a set of priests called a *quorum*. A ballot succeeded iff (if and only if) every priest in the quorum voted for the decree. Formally, a ballot B consisted of the following four components. (Unless otherwise qualified, *set* is taken to mean *finite set*.[6])

B_{dec} A decree (the one being voted on).

B_{qrm} A nonempty set of priests (the ballot's quorum).

B_{vot} A set of priests (the ones who cast votes for the decree).[7]

B_{bal} A ballot number.

A ballot B was said to be *successful* iff $B_{qrm} \subseteq B_{vot}$, so a successful ballot was one in which every quorum member voted.

Ballot numbers were chosen from an unbounded ordered set of numbers. If $B'_{bal} > B_{bal}$, then ballot B' was said to be *later* than ballot B. However, this indicated nothing about the order in which ballots were conducted; a later ballot could actually have taken place before an earlier one.

Paxon mathematicians defined three conditions on a set $\mathcal{B}$ of ballots, and then showed that consistency was guaranteed and progress was possible if the set of ballots that had taken place satisfied those conditions. The first two conditions were simple; they can be stated informally as follows.

$B1(\mathcal{B})$ Each ballot in $\mathcal{B}$ has a unique ballot number.

$B2(\mathcal{B})$ The quorums of any two ballots in $\mathcal{B}$ have at least one priest in common.

The third condition was more complicated. One Paxon manuscript contained the following, rather confusing, statement of it.

$B3(\mathcal{B})$ For every ballot B in $\mathcal{B}$, if any priest in B's quorum voted in an earlier ballot in $\mathcal{B}$, then the decree of B equals the decree of the latest of those earlier ballots.

Interpretation of this cryptic text was aided by the manuscript pictured in Figure 1, which illustrates condition $B3(\mathcal{B})$ with a set $\mathcal{B}$ of five ballots for a Synod consisting of the five priests A, B, Γ, Δ, and E. This set $\mathcal{B}$ contains five ballots, where for each ballot, the set of voters is the subset of the priests in the quorum whose names are enclosed in boxes. For example, ballot number 14 has decree α, a quorum containing three priests, and a set of two voters. Condition $B3(\mathcal{B})$ has the form "for every B in $\mathcal{B}$: ...", where "..." is a condition on ballot B. The conditions for the five ballots B of Figure 1 are as follows.

[5] Like some modern nations, Paxos had not fully grasped the nature of Athenian democracy.

[6] Although Paxon mathematicians were remarkably advanced for their time, they obviously had no knowledge of set theory. I have taken the liberty of translating the Paxon's more primitive notation into the language of modern set theory.

[7] Only priests in the quorum actually voted, but Paxon mathematicians found it easier to convince people that the protocol was correct if, in their proof, they allowed any priest to vote in any ballot.

#	*decree*	*quorum and* $\boxed{\textit{voters}}$				
2	α	A	B	Γ	$\boxed{\Delta}$	
5	β	A	B	$\boxed{\Gamma}$		E
14	α		$\boxed{\text{B}}$		Δ	$\boxed{\text{E}}$
27	β	$\boxed{\text{A}}$		$\boxed{\Gamma}$	$\boxed{\Delta}$	
29	β		$\boxed{\text{B}}$	Γ	Δ	

Fig. 1. Paxon manuscript showing a set $\mathcal{B}$, consisting of five ballots, that satisfies conditions $B1(\mathcal{B})$–$B3(\mathcal{B})$. (Explanatory column headings have been added.)

2. Ballot number 2 is the earliest ballot, so the condition on that ballot is trivially true.
5. None of ballot 5's four quorum members voted in an earlier ballot, so the condition on ballot 5 is also trivially true.
14. The only member of ballot 14's quorum to vote in an earlier ballot is Δ, who voted in ballot number 2, so the condition requires that ballot 14's decree must equal ballot 2's decree.
27. (This is a successful ballot.) The members of ballot 27's quorum are A, Γ, and Δ. Priest A did not vote in an earlier ballot, the only earlier ballot Γ voted in was ballot 5, and the only earlier ballot Δ voted in was ballot 2. The latest of these two earlier ballots is ballot 5, so the condition requires that ballot 27's decree must equal ballot 5's decree.
29. The members of ballot 29's quorum are B, Γ, and Δ. The only earlier ballot that B voted in was number 14, priest Γ voted in ballots 5 and 27, and Δ voted in ballots 2 and 27. The latest of these four earlier ballots is number 27, so the condition requires that ballot 29's decree must equal ballot 27's decree.

To state $B1(\mathcal{B})$–$B3(\mathcal{B})$ formally requires some more notation. A *vote* v was defined to be a quantity consisting of three components: a priest v_{pst}, a ballot number v_{bal}, and a decree v_{dec}. It represents a vote cast by priest v_{pst} for decree v_{dec} in ballot number v_{bal}. The Paxons also defined *null* votes to be votes v with $v_{bal} = -\infty$ and $v_{dec} = \text{BLANK}$, where $-\infty < b < \infty$ for any ballot number b, and BLANK is not a decree. For any priest p, they defined $null_p$ to be the unique null vote v with $v_{pst} = p$.

Paxon mathematicians defined a total ordering on the set of all votes, but part of the manuscript containing the definition has been lost. The remaining fragment indicates that, for any votes v and v', if $v_{bal} < v'_{bal}$ then $v < v'$. It is not known how the relative order of v and v' was defined if $v_{bal} = v'_{bal}$.

For any set $\mathcal{B}$ of ballots, the set $Votes(\mathcal{B})$ of votes in $\mathcal{B}$ was defined to consist of all votes v such that $v_{pst} \in B_{vot}$, $v_{bal} = B_{bal}$, and $v_{dec} = B_{dec}$ for some $B \in \mathcal{B}$. If p is a priest and b is either a ballot number or $\pm\infty$, then $MaxVote(b, p, \mathcal{B})$ was

defined to be the largest vote v in $Votes(\mathcal{B})$ cast by p with $v_{bal} < b$, or to be $null_p$ if there was no such vote. Since $null_p$ is smaller than any real vote cast by p, this means that $MaxVote(b, p, \mathcal{B})$ is the largest vote in the set

$$\{v \in Votes(\mathcal{B}) : (v_{pst} = p) \wedge (v_{bal} < b)\} \cup \{null_p\}$$

For any nonempty set Q of priests, $MaxVote(b, Q, \mathcal{B})$ was defined to equal the maximum of all votes $MaxVote(b, p, \mathcal{B})$ with p in Q.

Conditions $B1(\mathcal{B})$–$B3(\mathcal{B})$ are stated formally as follows.[8]

$$\begin{aligned}
B1(\mathcal{B}) &\triangleq \forall B, B' \in \mathcal{B} : (B \neq B') \Rightarrow (B_{bal} \neq B'_{bal}) \\
B2(\mathcal{B}) &\triangleq \forall B, B' \in \mathcal{B} : B_{qrm} \cap B'_{qrm} \neq \emptyset \\
B3(\mathcal{B}) &\triangleq \forall B \in \mathcal{B} : (MaxVote(B_{bal}, B_{qrm}, \mathcal{B})_{bal} \neq -\infty) \Rightarrow \\
&\qquad (B_{dec} = MaxVote(B_{bal}, B_{qrm}, \mathcal{B})_{dec})
\end{aligned}$$

Although the definition of *MaxVote* depends upon the ordering of votes, $B1(\mathcal{B})$ implies that $MaxVote(b, Q, \mathcal{B})_{dec}$ is independent of how votes with equal ballot numbers were ordered.

To show that these conditions imply consistency, the Paxons first showed that $B1(\mathcal{B})$–$B3(\mathcal{B})$ imply that, if a ballot B in $\mathcal{B}$ is successful, then any later ballot in $\mathcal{B}$ is for the same decree as B.

Lemma *If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then*

$$((B_{qrm} \subseteq B_{vot}) \wedge (B'_{bal} > B_{bal})) \Rightarrow (B'_{dec} = B_{dec})$$

for any B, B' in $\mathcal{B}$.

Proof of Lemma

For any ballot B in $\mathcal{B}$, let $\Psi(B, \mathcal{B})$ be the set of ballots in $\mathcal{B}$ later than B for a decree different from B's:

$$\Psi(B, \mathcal{B}) \triangleq \{B' \in \mathcal{B} : (B'_{bal} > B_{bal}) \wedge (B'_{dec} \neq B_{dec})\}$$

To prove the lemma, it suffices to show that if $B_{qrm} \subseteq B_{vot}$ then $\Psi(B, \mathcal{B})$ is empty. The Paxons gave a proof by contradiction. They assumed the existence of a B with $B_{qrm} \subseteq B_{vot}$ and $\Psi(B, \mathcal{B}) \neq \emptyset$, and obtained a contradiction as follows.[9]

1. Choose $C \in \Psi(B, \mathcal{B})$ such that $C_{bal} = \min\{B'_{bal} : B' \in \Psi(B, \mathcal{B})\}$.
   PROOF: C exists because $\Psi(B, \mathcal{B})$ is nonempty and finite.
2. $C_{bal} > B_{bal}$
   PROOF: By 1 and the definition of $\Psi(B, \mathcal{B})$.
3. $B_{vot} \cap C_{qrm} \neq \emptyset$
   PROOF: By $B2(\mathcal{B})$ and the hypothesis that $B_{qrm} \subseteq B_{vot}$.

---

[8] I use the Paxon mathematical symbol $\triangleq$, which meant *equals by definition*.

[9] Paxon mathematicians always provided careful, structured proofs of important theorems. They were not as sophisticated as modern mathematicians, who can omit many details and write paragraph-style proofs without ever making a mistake.

4. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} \geq B_{bal}$
   PROOF: By 2, 3 and the definition of $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$.
5. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B}) \in Votes(\mathcal{B})$
   PROOF: By 4 (which implies that $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$ is not a null vote) and the definition of $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$.
6. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{dec} = C_{dec}$.
   PROOF: By 5 and $B3(\mathcal{B})$.
7. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{dec} \neq B_{dec}$
   PROOF: By 6, 1, and the definition of $\Psi(B, \mathcal{B})$.
8. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} > B_{bal}$
   PROOF: By 4, since 7 and $B1(\mathcal{B})$ imply that $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} \neq B_{bal}$.
9. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B}) \in Votes(\Psi(B, \mathcal{B}))$
   PROOF: By 7, 8, and the definition of $\Psi(B, \mathcal{B})$.
10. $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})_{bal} < C_{bal}$
    PROOF: By definition of $MaxVote(C_{bal}, C_{qrm}, \mathcal{B})$.
11. Contradiction
    PROOF: By 9, 10, and 1.

End Proof of Lemma

With this lemma, it was easy to show that, if $B1$–$B3$ hold, then any two successful ballots are for the same decree.

THEOREM 1. *If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then*

$$((B_{qrm} \subseteq B_{vot}) \wedge (B'_{qrm} \subseteq B'_{vot})) \;\Rightarrow\; (B'_{dec} = B_{dec})$$

for any B, B' in $\mathcal{B}$.

Proof of Theorem
If $B'_{bal} = B_{bal}$, then $B1(\mathcal{B})$ implies $B' = B$. If $B'_{bal} \neq B_{bal}$, then the theorem follows immediately from the lemma.
End Proof of Theorem

The Paxons then proved a theorem asserting that if there are enough priests in the Chamber, then it is possible to conduct a successful ballot while preserving $B1$–$B3$. Although this does not guarantee progress, it at least shows that a balloting protocol based on $B1$–$B3$ will not deadlock.

THEOREM 2. *Let b be a ballot number and Q a set of priests such that $b > B_{bal}$ and $Q \cap B_{qrm} \neq \emptyset$ for all $B \in \mathcal{B}$. If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then there is a ballot B' with $B'_{bal} = b$ and $B'_{qrm} = B'_{vot} = Q$ such that $B1(\mathcal{B} \cup \{B'\})$, $B2(\mathcal{B} \cup \{B'\})$, and $B3(\mathcal{B} \cup \{B'\})$ hold.*

Proof of Theorem
Condition $B1(\mathcal{B} \cup \{B'\})$ follows from $B1(\mathcal{B})$, the choice of B'_{bal}, and the assumption about b. Condition $B2(\mathcal{B} \cup \{B'\})$ follows from $B2(\mathcal{B})$, the choice of B'_{qrm}, and the assumption about Q. If $MaxVote(b, Q, \mathcal{B})_{bal} = -\infty$ then let B'_{dec} be any decree, else let it equal $MaxVote(b, Q, \mathcal{B})_{dec}$. Condition $B3(\mathcal{B} \cup \{B'\})$ then follows from $B3(\mathcal{B})$.
End Proof of Theorem

2.2 The Preliminary Protocol

The Paxons derived the *preliminary protocol* from the requirement that conditions $B1(\mathcal{B})$–$B3(\mathcal{B})$ remain true, where $\mathcal{B}$ was the set of all ballots that had been or were being conducted. The definition of the protocol specified how the set $\mathcal{B}$ changed, but the set was never explicitly calculated. The Paxons referred to $\mathcal{B}$ as a quantity observed only by the gods, since it might never be known to any mortal.

Each ballot was initiated by a priest, who chose its number, decree, and quorum. Each priest in the quorum then decided whether or not to vote in the ballot. The rules determining how the initiator chose a ballot's number, decree, and quorum, and how a priest decided whether or not to vote in a ballot were derived directly from the need to maintain $B1(\mathcal{B})$–$B3(\mathcal{B})$.

To maintain $B1$, each ballot had to receive a unique number. By remembering (with notes in his ledger) what ballots he had previously initiated, a priest could easily avoid initiating two different ballots with the same number. To keep different priests from initiating ballots with the same number, the set of possible ballot numbers was partitioned among the priests. While it is not known how this was done, an obvious method would have been to let a ballot number be a pair consisting of an integer and a priest, using a lexicographical ordering, where

$$(13,\ \Gamma\rho\alpha\check{\iota}) < (13,\ \Lambda\iota\nu\sigma\epsilon\check{\iota}) < (15,\ \Gamma\rho\alpha\check{\iota})$$

since Γ came before Λ in the Paxon alphabet. In any case, it is known that every priest had an unbounded set of ballot numbers reserved for his use.

To maintain $B2$, a ballot's quorum was chosen to contain a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ of priests. Initially, $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ just meant a simple majority. Later, it was observed that fat priests were less mobile and spent more time in the Chamber than thin ones, so a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ was taken to mean any set of priests whose total weight was more than half the total weight of all priests, rather than a simple majority of the priests. When a group of thin priests complained that this was unfair, actual weights were replaced with symbolic weights based on a priest's attendance record. The primary requirement for a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ was that any two sets containing a $\mu\alpha\delta\zeta\partial\omega\rho\iota\pi\check{\iota}\sigma\epsilon\tau$ of priests had at least one priest in common. To maintain $B2$, the priest initiating a ballot B chose B_{qrm} to be a majority set.

Condition $B3$ requires that if $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$ is not equal to BLANK, then a ballot with number b and quorum Q must have decree $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$. If $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$ equals BLANK, then the ballot can have any decree. To maintain $B3(\mathcal{B})$, before initiating a new ballot with ballot number b and quorum Q, a priest p had to find $MaxVote(b,\ Q,\ \mathcal{B})_{dec}$. To do this, p had to find $MaxVote(b,\ q,\ \mathcal{B})$ for each priest q in Q.

Recall that $MaxVote(b,\ q,\ \mathcal{B})$ is the vote with the largest ballot number less than b among all the votes cast by q, or $null_q$ if q did not vote in any ballot numbered less than b. Priest p obtains $MaxVote(b,\ q,\ \mathcal{B})$ from q by an exchange of messages. Therefore, the first two steps in the protocol for conducting a single ballot initiated by p are:[10]

[10]Priests p and q could be the same. For simplicity, the protocol is described with p sending messages to himself in this case. In reality, a priest could talk to himself without the use of messengers.

(1) Priest p chooses a new ballot number b and sends a *NextBallot*(b) message to some set of priests.

(2) A priest q responds to the receipt of a *NextBallot*(b) message by sending a *LastVote*(b, v) message to p, where v is the vote with the largest ballot number less than b that q has cast, or his null vote $null_q$ if q did not vote in any ballot numbered less than b.

Priest q must use notes in the back of his ledger to remember what votes he had previously cast.

When q sends the *LastVote*(b, v) message, v equals *MaxVote*(b, q, $\mathcal{B}$). But the set $\mathcal{B}$ of ballots changes as new ballots are initiated and votes are cast. Since priest p is going to use v as the value of *MaxVote*(b, q, $\mathcal{B}$) when choosing a decree, to keep $B3(\mathcal{B})$ true it is necessary that *MaxVote*(b, q, $\mathcal{B}$) not change after q has sent the *LastVote*(b, v) message. To keep *MaxVote*(b, q, $\mathcal{B}$) from changing, q must cast no new votes with ballot numbers between v_{bal} and b. By sending the *LastVote*(b, v) message, q is promising not to cast any such vote. (To keep this promise, q must record the necessary information in his ledger.)

The next two steps in the balloting protocol (begun in step 1 by priest p) are:

(3) After receiving a *LastVote*(b, v) message from every priest in some majority set Q, priest p initiates a new ballot with number b, quorum Q, and decree d, where d is chosen to satisfy $B3$. He then records the ballot in the back of his ledger and sends a *BeginBallot*(b, d) message to every priest in Q.

(4) Upon receipt of the *BeginBallot*(b, d) message, priest q decides whether or not to cast his vote in ballot number b. (He may not cast the vote if doing so would violate a promise implied by a *LastVote*(b', v') message he has sent for some other ballot.) If q decides to vote for ballot number b, then he sends a *Voted*(b, q) message to p and records the vote in the back of his ledger.

The execution of step 3 is considered to add a ballot B to $\mathcal{B}$, where $B_{bal} = b$, $B_{qrm} = Q$, $B_{vot} = \emptyset$ (no one has yet voted in this ballot), and $B_{dec} = d$. In step 4, if priest q decides to vote in the ballot, then executing that step is considered to change the set $\mathcal{B}$ of ballots by adding q to the set B_{vot} of voters in the ballot $B \in \mathcal{B}$.

A priest has the option not to vote in step 4, even if casting a vote would not violate any previous promise. In fact, all the steps in this protocol are optional. For example, a priest q can ignore a *NextBallot*(b) message instead of executing step 2. Failure to take an action can prevent progress, but it cannot cause any inconsistency because it cannot make $B1(\mathcal{B})$–$B3(\mathcal{B})$ false. Since the only effect not receiving a message can have is to prevent an action from happening, message loss also cannot cause inconsistency. Thus, the protocol guarantees consistency even if priests leave the chamber or messages are lost.

Receiving multiple copies of a message can cause an action to be repeated. Except in step 3, performing the action a second time has no effect. For example, sending several *Voted*(b, q) messages in step 4 has the same effect as sending just one. The repetition of step 3 is prevented by using the entry made in the back of the ledger when it is executed. Thus, the consistency condition is maintained even if a messenger delivers the same message several times.

Steps 1–4 describe the complete protocol for initiating a ballot and voting on it.

All that remains is to determine the results of the balloting and announce when a decree has been selected. Recall that a ballot is successful iff every priest in the quorum has voted. The decree of a successful ballot is the one chosen by the Synod. The rest of the protocol is:

(5) If p has received a *Voted*(b, q) message from every priest q in Q (the quorum for ballot number b), then he writes d (the decree of that ballot) in his ledger and sends a *Success*(d) message to every priest.

(6) Upon receiving a *Success*(d) message, a priest enters decree d in his ledger.

Steps 1–6 describe how an individual ballot is conducted. The preliminary protocol allows any priest to initiate a new ballot at any time. Each step maintains $B1(\mathcal{B})$–$B3(\mathcal{B})$, so the entire protocol also maintains these conditions. Since a priest enters a decree in his ledger only if it is the decree of a successful ballot, Theorem 1 implies that the priests' ledgers are consistent. The protocol does not address the question of progress.

In step 3, if the decree d is determined by condition $B3$, then it is possible that this decree is already written in the ledger of some priest. That priest need not be in the quorum Q; he could have left the Chamber. Thus, consistency would not be guaranteed if step 3 allowed any greater freedom in choosing d.

2.3 The Basic Protocol

In the preliminary protocol, a priest must record (i) the number of every ballot he has initiated, (ii) every vote he has cast, and (iii) every *LastVote* message he has sent. Keeping track of all this information would have been difficult for the busy priests. The Paxons therefore restricted the preliminary protocol to obtain the more practical *basic protocol* in which each priest p had to maintain only the following information in the back of his ledger:

lastTried$[p]$ The number of the last ballot that p tried to initiate, or $-\infty$ if there was none.

prevVote$[p]$ The vote cast by p in the highest-numbered ballot in which he voted, or $-\infty$ if he never voted.

nextBal$[p]$ The largest value of b for which p has sent a *LastVote*(b, v) message, or $-\infty$ if he has never sent such a message.

Steps 1–6 of the preliminary protocol describe how a single ballot is conducted by its initiator, priest p. The preliminary protocol allows p to conduct any number of ballots concurrently. In the basic protocol, he conducts only one ballot at a time—ballot number *lastTried*$[p]$. After p initiates this ballot, he ignores messages that pertain to any other ballot that he had previously initiated. Priest p keeps all information about the progress of ballot number *lastTried*$[p]$ on a slip of paper. If he loses that slip of paper, then he stops conducting the ballot.

In the preliminary protocol, each *LastVote*(b, v) message sent by a priest q represents a promise not to vote in any ballot numbered between v_{bal} and b. In the basic protocol, it represents the stronger promise not to cast a new vote in any ballot numbered less than b. This stronger promise might prevent him from casting a vote in step 4 of the basic protocol that he would have been allowed to cast in the preliminary protocol. However, since the preliminary protocol always gives q

the option of not casting his vote, the basic protocol does not require him to do anything not allowed by the preliminary protocol.

Steps 1–6 of the preliminary protocol become the following six steps for conducting a ballot in the basic protocol. (All information used by p to conduct the ballot, other than $lastTried[p]$, $prevVote[p]$, and $nextBal[p]$, is kept on a slip of paper.)

(1) Priest p chooses a new ballot number b greater than $lastTried[p]$, sets $lastTried[p]$ to b, and sends a $NextBallot(b)$ message to some set of priests.

(2) Upon receipt of a $NextBallot(b)$ message from p with $b > nextBal[q]$, priest q sets $nextBal[q]$ to b and sends a $LastVote(b, v)$ message to p, where v equals $prevVote[q]$. (A $NextBallot(b)$ message is ignored if $b \leq nextBal[q]$.)

(3) After receiving a $LastVote(b, v)$ message from every priest in some majority set Q, where $b = lastTried[p]$, priest p initiates a new ballot with number b, quorum Q, and decree d, where d is chosen to satisfy $B3$. He then sends a $BeginBallot(b, d)$ message to every priest in Q.

(4) Upon receipt of a $BeginBallot(b, d)$ message with $b = nextBal[q]$, priest q casts his vote in ballot number b, sets $prevVote[q]$ to this vote, and sends a $Voted(b, q)$ message to p. (A $BeginBallot(b, d)$ message is ignored if $b \neq nextBal[q]$.)

(5) If p has received a $Voted(b, q)$ message from every priest q in Q (the quorum for ballot number b), where $b = lastTried[p]$, then he writes d (the decree of that ballot) in his ledger and sends a $Success(d)$ message to every priest.

(6) Upon receiving a $Success(d)$ message, a priest enters decree d in his ledger.

The basic protocol is a restricted version of the preliminary protocol, meaning that every action allowed by the basic protocol is also allowed by the preliminary protocol. Since the preliminary protocol satisfies the consistency condition, the basic protocol also satisfies that condition. Like the preliminary protocol, the basic protocol does not require that any action ever be taken, so it does not addresses the question of progress.

The derivation of the basic protocol from $B1$–$B3$ made it obvious that the consistency condition was satisfied. However, some similarly "obvious" ancient wisdom had turned out to be false, and skeptical citizens demanded a more rigorous proof. Their Paxon mathematicians' proof that the protocol satisfies the consistency condition is reproduced in the appendix.

2.4 The Complete Synod Protocol

The basic protocol maintains consistency, but it cannot ensure any progress because it states only what a priest may do; it does not require him to do anything. The complete protocol consists of the same six steps for conducting a ballot as the basic protocol. To help achieve progress, it includes the obvious additional requirement that priests perform steps 2–6 of the protocol as soon as possible. However, to meet the progress condition, it is necessary that some priest be required to perform step 1, which initiates a ballot. The key to the complete protocol lay in determining when a priest should initiate a ballot.

Never initiating a ballot will certainly prevent progress. However, initiating too may ballots can also prevent progress. If b is larger than any other ballot number, then the receipt of a $NextBallot(b)$ message by priest q in step 2 may elicit a

promise that prevents him from voting in step 4 for any previously initiated ballot. Thus, the initiation of a new ballot can prevent any previously initiated ballot from succeeding. If new ballots are continually initiated with increasing ballot numbers before the previous ballots have a chance to succeed, then no progress might be made.

Achieving the progress condition requires that new ballots be initiated until one succeeds, but that they not be initiated too frequently. To develop the complete protocol, the Paxons first had to know how long it took messengers to deliver messages and priests to respond. They determined that a messenger who did not leave the Chamber would always deliver a message within 4 minutes, and a priest who remained in the Chamber would always perform an action within 7 minutes of the event that caused the action.[11] Thus, if p and q were in the Chamber when some event caused p to send a message to q, and q responded with a reply to p, then p would receive that reply within 22 minutes if neither messenger left the Chamber. (Priest p would send the message within 7 minutes of the event, q would receive the message within 4 more minutes, he would respond within 7 minutes, and the reply would reach p within 4 more minutes.)

Suppose that only a single priest p was initiating ballots, and that he did so by sending a message to every priest in step 1 of the protocol. If p initiated a ballot when a majority set of priests was in the chamber, then he could expect to execute step 3 within 22 minutes of initiating the ballot, and to execute step 5 within another 22 minutes. If he was unable to execute the steps by those times, then either some priest or messenger left the Chamber after p initiated the ballot, or a larger-numbered ballot had previously been initiated by another priest (before p became the only priest to initiate ballots). To handle the latter possibility, p had to learn about any ballot numbers greater than $lastTried[p]$ used by other priests. This could be done by extending the protocol to require that if a priest q received a $NextBallot(b)$ or a $BeginBallot(b, d)$ message from p with $b < nextBal[q]$, then he sent p a message containing $nextBal[q]$. Priest p would then initiate a new ballot with a larger ballot number.

Still assuming that p was the only priest initiating ballots, suppose that he were required to initiate a new ballot iff (i) he had not executed step 3 or step 5 within the previous 22 minutes, or (ii) he learned that another priest had initiated a higher-numbered ballot. If the Chamber doors were locked with p and a majority set of priests inside, then a decree would be passed and recorded in the ledgers of all priests in the Chamber within 99 minutes. (It could take 22 minutes for p to start the next ballot, 22 more minutes to learn that another priest had initiated a larger-numbered ballot, then 55 minutes to complete steps 1–6 for a successful ballot.) Thus, the progress condition would be met if only a single priest, who did not leave the chamber, were initiating ballots.

The complete protocol therefore included a procedure for choosing a single priest, called the *president*, to initiate ballots. In most forms of government, choosing a

[11] I am assuming a value of 30 seconds for the $\delta\zeta\partial\iota\phi\check{\iota}$, the Paxon unit of time. This value is within the range determined from studies of hourglass shards. The reaction time of priests was so long because they had to respond to every message within 7 minutes (14 $\delta\zeta\partial\iota\phi\check{\iota}$), even if a number of messages arrived simultaneously.

president can be a difficult problem. However, the difficultly arises only because most governments require that there be exactly one president at any time. In the United States, for example, chaos would result if some people thought Bush had been elected president while others thought that Dukakis had, since one of them might decide to sign a bill into law while the other decided to veto it. However, in the Paxon Synod, having multiple presidents could only impede progress; it could not cause inconsistency. For the complete protocol to satisfy the progress condition, the method for choosing the president needed only to satisfy the following *presidential selection requirement*:

> If no one entered or left the Chamber, then after T minutes exactly one priest in the Chamber would consider himself to be the president.

If the presidential selection requirement were met, then the complete protocol would have the property that if a majority set of priests were in the chamber and no one entered or left the Chamber for $T + 99$ minutes, then at the end of that period every priest in the Chamber would have a decree written in his ledger.

The Paxons chose as president the priest whose name was last in alphabetical order among the names of all priests in the Chamber, though we don't know exactly how this was done. The presidential selection requirement would have been satisfied if a priest in the Chamber sent a message containing his name to every other priest at least once every $T - 11$ minutes, and a priest considered himself to be president iff he received no message from a "higher-named" priest for T minutes.

The complete Synod protocol was obtained from the basic protocol by requiring priests to perform steps 2–6 promptly, adding a method for choosing a president who initiated ballots, and requiring the president to initiate ballots at the appropriate times. Many details of the protocol are not known. I have described simple methods for selecting a president and for deciding when the president should initiate a new ballot, but they are undoubtedly not the ones used in Paxos. The rules I have given require the president to keep initiating ballots even after a decree has been chosen, thereby ensuring that priests who have just entered the Chamber learn about the chosen decree. There were obviously better ways to make sure priests learned about the decree after it had been chosen. Also, in the course of selecting a president, each priest probably sent his value of $lastTried[p]$ to the other priests, allowing the president to choose a large enough ballot number on his first try.

The Paxons realized that any protocol to achieve the progress condition must involve measuring the passage of time.[12] The protocols given above for selecting a president and initiating ballots are easily formulated as precise algorithms that set timers and perform actions when time-outs occur—assuming perfectly accurate timers. A closer analysis reveals that such protocols can be made to work with timers having a known bound on their accuracy. The skilled glass blowers of Paxos had no difficulty constructing suitable hourglass timers.

Given the sophistication of Paxon mathematicians, it is widely believed that they must have found an optimal algorithm to satisfy the presidential selection requirement. We can only hope that this algorithm will be discovered in future

[12]However, many centuries were to pass before a rigorous proof of this result was given.[Fischer et al. 1985]

excavations on Paxos.

3 The Multi-Decree Parliament

When Parliament was established, a protocol to satisfy its consistency and progress requirements was derived from the Synod protocol. The derivation and properties of the original parliamentary protocol are described in Sections 3.1 and 3.2. Section 3.3 discusses the further evolution of the protocol.

3.1 The Protocol

Instead of passing just one decree, the Paxon Parliament had to pass a series of numbered decrees. As in the Synod protocol, a president was elected. Anyone who wanted a decree passed would inform the president, who would assign a number to the decree and attempt to pass it. Logically, the parliamentary protocol used a separate instance of the complete Synod protocol for each decree number. However, a single president was selected for all these instances, and he performed the first two steps of the protocol just once.

The key to deriving the parliamentary protocol is the observation that, in the Synod protocol, the president does not choose the decree or the quorum until step 3. A newly elected president p can send to some set of legislators a single message that serves as the $NextBallot(b)$ message for all instances of the Synod protocol. (There are an infinite number of instances—one for each decree number.) A legislator q can reply with a single message that serves as the *LastVote* messages for step 2 of all instances of the Synod protocol. This message contains only a finite amount of information, since q can have voted in only a finite number of instances.

When the new president has received a reply from every member of a majority set, he is ready to perform step 3 for every instance of the Synod protocol. For some finite number of instances (decree numbers), the choice of decree in step 3 will be determined by $B3$. The president immediately performs step 3 for each of those instances to try passing these decrees. Then, whenever he receives a request to pass a decree, he chooses the lowest-numbered decree that he is still free to choose, and he performs step 3 for that decree number (instance of the Synod protocol) to try to pass the decree.

The following modifications to this simple protocol lead to the actual Paxon Parliament's protocol.

—There is no reason to go through the Synod protocol for a decree number whose outcome is already known. Therefore, if a newly elected president p has all decrees with numbers less than or equal to n written in his ledger, then he sends a $NextBallot(b, n)$ message that serves as a $NextBallot(b)$ message in all instances of the Synod protocol for decree numbers larger than n. In his response to this message, legislator q informs p of all decrees numbered greater than n that already appear in q's ledger (in addition to sending the usual *LastVote* information for decrees not in his ledger), and he asks p to send him any decrees numbered n or less that are not in his ledger.

—Suppose decrees 125 and 126 are introduced late Friday afternoon, decree 126 is passed and is written in one or two ledgers, but before anything else happens, the legislators all go home for the weekend. Suppose also that the following Monday,

$\Delta\phi\omega\rho\kappa$ is elected the new president and learns about decree 126, but she has no knowledge of decree 125 because the previous president and all legislators who had voted for it are still out of the Chamber. She will hold a ballot that passes decree 126, which leaves a gap in the ledgers. Assigning number 125 to a new decree would cause it to appear earlier in the ledger than decree 126, which had been passed the previous week. Passing decrees out of order in this way might cause confusion—for example, if the citizen who proposed the new decree did so because he knew decree 126 had already passed. Instead, $\Delta\phi\omega\rho\kappa$ would attempt to pass

125: *The ides of February is national olive day*

a traditional decree that made absolutely no difference to anyone in Paxos. In general, a new president would fill any gaps in his ledger by passing the "olive-day" decree.

The consistency and progress properties of the parliamentary protocol follow immediately from the corresponding properties of the Synod protocol from which it was derived. To our knowledge, the Paxons never bothered writing a precise description of the parliamentary protocol because it was so easily derived from the Synod protocol.

3.2 Properties of the Protocol

3.2.1 *The Ordering of Decrees* Balloting could take place concurrently for many different decree numbers, with ballots initiated by different legislators—each thinking he was president when he initiated the ballot. We cannot say precisely in what order decrees would be passed, especially without knowing how a president was selected. However, there is one important property about the ordering of decrees that can be deduced.

A decree was said to to be *proposed* when it was chosen by the president in step 3 of the corresponding instance of the Synod protocol. The decree was said to be *passed* when it was written for the first time in a ledger. Before a president could propose any new decrees, he had to learn from all the members of a majority set what decrees they had voted for. Any decree that had already been passed must have been voted for by at least one legislator in the majority set. Therefore, the president must have learned about all previously passed decrees before initiating any new decree. The president would not fill a gap in the ledgers with an important decree—that is, with any decree other than the "olive-day" decree. He would also not propose decrees out of order. Therefore, the protocol satisfied the following *decree-ordering property*.

> If decrees A and B are important and decree A was passed before decree B was proposed, then A has a lower decree number than B.

3.2.2 *Behind Closed Doors* Although we don't know the details involved in choosing a new president, we do know exactly how Parliament functioned when the president had been chosen and no one was entering or leaving the Chamber. Upon receiving a request to pass a decree—either directly from a citizen or relayed from another legislator—the president assigned the decree a number and passed it with the following exchange of messages. (The numbers refer to the corresponding steps in the Synod protocol.)

(3) The president sent a *BeginBallot* message to each legislator in a quorum.
(4) Each legislator in the quorum sent a *Voted* message to the president.
(5) The president sent a *Success* message to every legislator.

This is a total of three message delays and about $3N$ messages, assuming a parliament of N legislators and a quorum of about $N/2$. Moreover, if Parliament was busy, the president would combine the *BeginBallot* message for one decree with the *Success* message for a previous one, for a total of only $2N$ messages per decree.

3.3 Further Developments

Governing the island turned out to be a more complex task than the Paxons realized. A number of problems arose whose solutions required changes to the protocol. The most important of these changes are described below.

3.3.1 *Picking a President* The president of parliament was originally chosen by the method that had been used in the Synod, which was based purely on the alphabetical ordering of names. Thus, when legislator $\Omega\kappa\iota$ returned from a six-month vacation, he was immediately made president—even though he had no idea what had happened in his absence. Parliamentary activity came to a halt while $\Omega\kappa\iota$, who was a slow writer, laboriously copied six months worth of decrees to bring his ledger up to date.

This incident led to a debate about the best way to choose a president. Some Paxons urged that once a legislator became president, he should remain president until he left the Chamber. An influential group of citizens wanted the richest legislator in the Chamber to be president, since he could afford to hire more scribes and other servants to help him with the presidential duties. They argued that once a rich legislator had brought his ledger up to date, there was no reason for him not to assume the presidency. Others, however, argued that the most upstanding citizen should be made president, regardless of wealth. Upstanding probably meant less likely to be dishonest, although no Paxon would publicly admit the possibility of official malfeasance. Unfortunately, the outcome of this debate is not known; no record exists of the presidential selection protocol that was ultimately used.

3.3.2 *Long Ledgers* As the years progressed and Parliament passed more and more decrees, Paxons had to pore over an ever longer list of decrees to find the current olive tax or what color goat could be sold. A legislator who returned to the Chamber after an extended voyage had to do quite a bit of copying to bring his ledger up to date. Eventually, the legislators were forced to convert their ledgers from lists of decrees into law books that contained only the current state of the law and the number of the last decree whose passage was reflected in that state.

To learn the current olive tax, one looked in the law book under "taxes"; to learn what color goat could be sold, one looked under "mercantile law". If a legislator's ledger contained the law through decree 1298 and he learned that decree 1299 set the olive tax to 6 drachmas per ton, he just changed the entry for the olive-tax law and noted that his ledger was complete through decree 1299. If he then learned about decree 1302, he would write it down in the back of the ledger and wait until he learned about decrees 1300 and 1301 before incorporating decree 1302 into the law book.

To enable a legislator who had been gone for a short time to catch up without copying the entire law book, legislators kept a list of the past week's decrees in the back of the book. They could have kept this list on a slip of paper, but it was convenient for a legislator to enter decrees in the back of the ledger as they were passed and update the law book only two or three times a week.

3.3.3 *Bureaucrats* As Paxos prospered, legislators became very busy. Parliament could no longer handle all details of government, so a bureaucracy was established. Instead of passing a decree to declare whether each lot of cheese was fit for sale, Parliament passed a decree appointing a cheese inspector to make those decisions.

It soon became evident that selecting bureaucrats was not as simple as it first seemed. Parliament passed a decree making *Δĭκστρα* the first cheese inspector. After some months, merchants complained that *Δĭκστρα* was too strict and was rejecting perfectly good cheese. Parliament then replaced him by passing the decree

1375: *Γωυδα is the new cheese inspector*

But *Δĭκστρα* did not pay close attention to what Parliament did, so he did not learn of this decree right away. There was a period of confusion in the cheese market when both *Δĭκστρα* and *Γωυδα* were inspecting cheese and making conflicting decisions.

To prevent such confusion, the Paxons had to guarantee that a position could be held by at most one bureaucrat at any time. To do this, a president included as part of each decree the time and date when it was proposed. A decree making *Δĭκστρα* the cheese inspector might read

2716: *8:30 15 Jan 72—Δĭκστρα is cheese inspector for 3 months*

This declares his term to begin either at 8:30 on 15 January or when the previous inspector's term ended—whichever was later. His term would end at 8:30 on 15 March, unless he explicitly resigned by asking the president to pass a decree like

2834: *9:15 3 Mar 72—Δĭκστρα resigns as cheese inspector*

A bureaucrat was appointed for a short term, so he could be replaced quickly—for example, if he left the island. Parliament would pass a decree to extend the bureaucrat's term if he was doing a satisfactory job.

A bureaucrat needed to tell time to determine if he currently held a post. Mechanical clocks were unknown on Paxos, but Paxons could tell time accurately to within 15 minutes by the position of the sun or the stars.[13] If *Δĭκστρα*'s term began at 8:30, he would not start inspecting cheese until his celestial observations indicated that it was 8:45.

It is easy to make this method of appointing bureaucrats work if higher-numbered decrees always have later proposal times. But what if Parliament passed the decrees

2854: *9:45 9 Apr 78—Φρανσεζ is wine taster for 2 months*

2855: *9:20 9 Apr 78—Πνυελĭ is wine taster for 1 month*

[13]Cloudy days are rare in Paxos's balmy climate.

that were proposed between 9:30 and 9:35 by different legislators who both thought they were president? Such out-of-order proposal times are easily prevented because the parliamentary protocol satisfies the following property.

> If two decrees are passed by different presidents, then one of the presidents proposed his decree after learning that the other decree had been proposed.

To see that this property is satisfied, suppose that ballot number b was successful for decree D, ballot number b' was successful for decree D', and $b < b'$. Let q be a legislator who voted in both ballots. The balloting for D' began with a $NextBallot(b', n)$ message. If the sender of that message did not already know about D, then n is less than the decree number of D, and q's reply to the *NextBallot* message must state that he voted for D.

3.3.4 *Learning the Law* In addition to requesting the passage of decrees, ordinary citizens needed to inquire about the current law of the land. The Paxons at first thought that a citizen could simply examine the ledger of any legislator, but the following incident demonstrated that a more sophisticated approach was needed. For centuries, it had been legal to sell only white goats. A farmer named *Δωλεφ* got Parliament to pass the decree

77: *The sale of black goats is permitted*

Δωλεφ then instructed his goatherd to sell some black goats to a merchant named *Σκεεν*. As a law-abiding citizen, *Σκεεν* asked legislator *Σπωκμεĭρ* if such a sale would be legal. But *Σπωκμεĭρ* had been out of the Chamber and had no entry in his ledger past decree 76. He advised *Σκεεν* that the sale would be illegal under the current law, so *Σκεεν* refused to buy the goats.

This incident led to the formulation of the following *monotonicity condition* on inquiries about the law.

> If one inquiry precedes a second inquiry, then the second inquiry cannot reveal an earlier state of the law than the first.

If a citizen learns that a particular decree has been passed, then the process of acquiring that knowledge is considered to be an implicit inquiry to which this condition applies. As we will see, the interpretation of the monotonicity condition changed over the years.

Initially, the monotonicity condition was achieved by passing a decree for each inquiry. If *Σδνĭδερ* wanted to know the current tax on olives, he would get Parliament to pass a decree such as

87: *Citizen Σδνĭδερ is reading the law*

He would then read any ledger complete at least through decree 86 to learn the olive tax as of that decree. If citizen *Γρεες* then inquired about the olive tax, the decree for his inquiry was proposed after decree 87 was passed, so the decree-ordering property (Section 3.2.1) implies that it received a decree number greater than 87. Therefore, *Γρεες* could not obtain an earlier value of the olive tax than *Σδνĭδερ*.

This method of reading the law satisfied the monotonicity condition when *precedes* was interpreted to mean that inquiry A precedes inquiry B iff A finished at an earlier time than B began.

Passing a decree for every inquiry soon proved too cumbersome. The Paxons realized that a simpler method was possible if they weakened the monotonicity condition by changing the interpretation of *precedes*. They decided that for one event to precede another, the first event not only had to happen at an earlier time, but it had to be able to causally affect the second event. The weaker monotonicity condition prevents the problem first encountered by farmer *Δωλεφ* and merchant *Σκεεν* because there is a causal chain of events between the end of the implicit inquiry by *Δωλεφ* and the beginning of the inquiry by *Σκεεν*.

The weaker monotonicity condition was met by using decree numbers in all business transactions and inquiries. For example, farmer *Δωλεφ*, whose flock included many nonwhite goats, got Parliament to pass the decree

277: *The sale of brown goats is permitted*

When selling his brown goats to *Σκεεν*, he informed the merchant that the sale was legal as of decree number 277. *Σκεεν* then asked legislator *Σπωκμεĭρ* if the sale were legal under the law through at least decree 277. If *Σπωκμεĭρ*'s ledger was not complete through decree 277, he would either wait until it was or else tell *Σκεεν* to ask someone else. If *Σπωκμεĭρ*'s ledger went through decree 298, then he would tell *Σκεεν* that the sale was legal as of decree number 298. Merchant *Σκεεν* would remember the number 298 for use in his next business transaction or inquiry about the law.

The Paxons had satisfied the monotonicity condition, but ordinary citizens disliked having to remember decree numbers. Again, the Paxons solved the problem by re-interpreting the monotonicity condition—this time, by changing the meaning of *state of the law*. They divided the law into separate areas, and a legislator was chosen as specialist for each area. The current state of each area of the law was determined by that specialist's ledger. For example, suppose decree 1517 changed the tariff law and decree 1518 changed the tax law. The tax law would change first if the tax-law specialist learned of both decrees before the tariff-law specialist learned of either, yielding a state of the law that could not be obtained by enacting the decrees in numerical order.

To avoid conflicting definitions of the current state, the Paxons required that there be at most one specialist at a time for any area. This requirement was satisfied by using the same method to choose specialists that was used to choose bureaucrats (see Section 3.3.3). If each inquiry involved only a single area of the law, monotonicity was then achieved by directing the inquiry to that area's specialist, who answered it from his ledger. Since learning that a law had passed constituted the result of an implicit inquiry, the Paxons required that a decree change at most one area of the law, and that notification of the decree's passage could come only from the area's specialist.

Inquiries involving multiple areas were not hard to handle. When merchant *Λισκωφ* asked if the tariff on an imported golden fleece was higher than the sales tax on one purchased locally, the tax-law and tariff-law specialists had to cooperate to provide an answer. For example, the tax specialist could answer *Λισκωφ* by first

asking the tariff specialist for the tariff on golden fleeces, so long as he made no changes to his ledger before receiving a reply.

This method proved satisfactory until it became necessary to make a sweeping change to several areas of the law at one time. The Paxons then realized that the necessary requirement for maintaining monotonicity was not that a decree affect only a single area, but that every area it affects have the same specialist. Parliament could change several areas of the law with a single decree by first appointing a single legislator to be the specialist for all those areas. Moreover, the same area could have multiple specialists, so long as that area of the law was not allowed to change. Just before income taxes were due, Parliament would appoint several tax-law specialists to handle the seasonal flood of inquiries about the tax law.

3.3.5 *Dishonest Legislators and Honest Mistakes* Despite official assertions to the contrary, there must have been a few dishonest legislators in the history of Paxos. When caught, they were probably exiled. By sending contradictory messages, a malicious legislator could cause different legislators' ledgers to be inconsistent. Inconsistency could also result from a lapse of memory by an honest legislator or messenger.

When inconsistencies were recognized, they could easily be corrected by passing decrees. For example, disagreement about the current olive tax could be eliminated by passing a new decree declaring the tax to have a certain value. The difficult problem lay in correcting inconsistent ledgers even if no one was aware of the inconsistency.

The existence of dishonesty or mistakes by legislators can be inferred from the redundant decrees that began appearing in ledgers several years after the founding of Parliament. For example, the decree

2605: *The olive tax is 9 drachmas per ton*

was passed even though decree 2155 had already set the olive tax to 9 drachmas per ton, and no intervening decree had changed it. Parliament apparently cycled through its laws every six months so that even if legislators' ledgers were initially inconsistent, all legislators would agree on the current law of the land within six months. It is believed that by the use of these redundant decrees, the Paxons made their Parliament self-stabilizing. (*Self-stabilizing* is a modern term due to Dijkstra [Dijkstra 1974].)

It is not clear precisely what self-stabilization meant in a Parliament with legislators coming and going at will. The Paxons would not have been satisfied with a definition that required all legislators to be in the Chamber at one time before consistency could be guaranteed. However, achieving consistency required that if one legislator had an entry in his ledger for a certain decree number and another did not, then the second legislator would eventually fill in that entry.

Unfortunately, we don't know exactly what sort of self-stabilization property the Paxon Parliament possessed or how it was achieved. Paxon mathematicians undoubtedly addressed the problem, but their work has not yet been found. I hope that future archaeological expeditions to Paxos will give high priority to the search for manuscripts on self-stabilization.

3.3.6 *Choosing New Legislators* At first, membership in Parliament was hereditary, passing from parent to child. When the elder statesman *Παρνας* retired, he gave his ledger to his son, who carried on without interruption. It made no difference to other legislators which *Παρνας* they communicated with.

As old families emigrated and new ones immigrated, this system had to change. The Paxons decided to add and remove members of Parliament by decree. This posed a circularity problem: membership in Parliament was determined by which decrees were passed, but passing a decree required knowing what constituted a majority set, which in turn depended upon who was a member of Parliament. The circularity was broken by letting the membership of Parliament used in passing decree n be specified by the law as of decree $n - 3$. A president could not try to pass decree 3255 until he knew all decrees through decree 3252. In practice, after passing the decree

3252: *Στρωνγ is now a legislator*

the president would immediately pass the "olive-day" decree as decrees 3253 and 3254.

Changing the composition of Parliament in this way was dangerous and had to be done with care. The consistency and progress conditions would always hold. However, the progress condition guaranteed progress only if a majority set was in the Chamber; it did not guarantee that a majority set would ever be there. In fact, the mechanism for choosing legislators led to the downfall of the Parliamentary system in Paxos. Because of a scribe's error, a decree that was supposed to honor sailors who had drowned in a shipwreck instead declared them to be the only members of Parliament. Its passage prevented any new decrees from being passed—including the decrees proposed to correct the mistake. Government in Paxos came to a halt. A general named *Λαμπσων* took advantage of the confusion to stage a coup, establishing a military dictatorship that ended centuries of progressive government. Paxos grew weak under a series of corrupt dictators, and was unable to repel an invasion from the east that led to the destruction of its civilization.

4 Relevance to Computer Science

4.1 The State Machine Approach

Although Paxos's Parliament was destroyed many centuries ago, its protocol is still useful. For example, consider a simple distributed database system that might be used as a name server. A state of the database consists of an assignment of values to names. Copies of the database are maintained by multiple servers. A client program can issue, to any server, a request to read or change the value assigned to a name. There are two kinds of read request: a *slow read*, which returns the value currently assigned to a name, and a *fast read*, which is faster but might not reflect a recent change to the database.

There is an obvious correspondence between this database system and the Paxon Parliament:

Parliament		Distributed Database
legislator	$\leftrightarrow$	server
citizen	$\leftrightarrow$	client program
current law	$\leftrightarrow$	database state

command:	**read**(*name*, *client*)	**update**(*name*, *val*, *client*)
response:	(*client*, value of *name*)	(*client*, "ok")
new state:	Same as current state	Same as current state except value of *name* changed to *val*

Fig. 2. State machine for simple database.

A client's request to change a value is performed by passing a decree. A *slow read* involves passing a decree, as described in Section 3.3.4. A *fast read* is performed by reading the server's current version of the database. The Paxon Parliament protocol provides a distributed, fault-tolerant implementation of the database system,

This method of implementing a distributed database is an instance of the state machine approach, first proposed in [Lamport 1978]. In this approach, one first defines a *state machine*, which consists of a set of states, a set of commands, a set of responses, and a function that assigns a response/state pair (a pair consisting of a response and a state) to each command/state pair. Intuitively, a state machine executes a command by producing a response and changing its state; the command and the machine's current state determine its response and its new state. For the distributed database, a state-machine state is just a database state. The state-machine commands and the function specifying the response and new state are described in Figure 2.

In the state-machine approach, a system is implemented with a network of server processes. The servers transform client requests into state machine commands, execute the commands, and transform the state-machine responses into replies to clients. A general algorithm ensures that all servers obtain the same sequence of commands, thereby ensuring that they all produce the same sequence of responses and state changes—assuming they all start from the same initial state. In the database example, a client request to perform a *slow read* or to change a value is transformed into a state-machine **read** or **update** command. That command is executed, and the state-machine response is transformed into a reply to the client, which is sent to him by the server who received his request. Since all servers perform the same sequence of state-machine commands, they all maintain consistent versions of the database. However, at any time, some servers may have earlier versions than others because a state-machine command need not be executed at the same time by all servers. A server uses his current version of the state to reply to a *fast read* request, without executing a state-machine command.

The functionality of the system is expressed by the state machine, which is just a function from command/state pairs to response/state pairs. Problems of synchronization and fault-tolerance are handled by the general algorithm with which servers obtain the sequence of commands. When designing a new system, only the state machine is new. The servers obtain the state-machine commands by a standard distributed algorithm that has already been proved correct. Functions are much easier to design, and to get right, than distributed algorithms.

The first algorithm for implementing an arbitrary state machine appeared in [Lamport 1978]. Later, algorithms were devised to tolerate up to any fixed number f of arbitrary failures [Lamport 1984]. These algorithms guarantee that, if fewer

than f processes fail, then state machine commands are executed within a fixed length of time. The algorithms are thus suitable for applications requiring real-time response.[14] But if more than f failures occur, then different servers may have inconsistent copies of the state machine. Moreover, the inability of two servers to communicate with each other is equivalent to the failure of one of them. For a system to have a low probability of losing consistency, it must use an algorithm with a large value of f, which in turn implies a large cost in redundant hardware, communication bandwidth, and response time.

The Paxon Parliament's protocol provides another way to implement an arbitrary state machine. The legislators' law book corresponds to the machine state, and passing a decree corresponds to executing a state-machine command. The resulting algorithm is less robust and less expensive than the earlier algorithms. It does not tolerate arbitrary, malicious failures, nor does it guarantee bounded-time response. However, consistency is maintained despite the (benign) failure of any number of processes and communication paths. The Paxon algorithm is suitable for systems with modest reliability requirements that do not justify the expense of an extremely fault-tolerant, real-time implementation.

If the state machine is executed with an algorithm that guarantees bounded-time response, then time can be made part of the state, and machine actions can be triggered by the passage of time. For example, consider a system for granting ownership of resources. The state can include the time at which a client was granted a resource, and the state machine can automatically execute a command to revoke ownership if the client has held the resource too long.

With the Paxon algorithm, time cannot be made part of the state in such a natural way. If failures occur, it can take arbitrarily long to execute a command (pass a decree), and one command can be executed before (appear earlier in the sequence of decrees than) another command that was issued earlier. However, a state machine can still use real time the same way the Paxon Parliament did. For example, the method described in Section 3.3.3 for deciding who was the current cheese inspector can be used to decide who is the current owner of a resource.

4.2 Commit Protocols

The Paxon Synod protocol is similar to standard three-phase commit protocols [Bernstein et al. 1987; Skeen 1982]. A Paxon ballot and a three-phase commit protocol both involve the exchange of five messages between a coordinator (the president) and the other quorum members (legislators). A commit protocol chooses one of two values—*commit* or *abort*—while the Synod protocol chooses an arbitrary decree. To convert a commit protocol to a Synod protocol, one sends the decree in the initial round of messages. A *commit* decision means that this decree was passed, and an *abort* decision means that the "olive-day" decree was passed.

The Synod protocol differs from a converted commit protocol because the decree is not sent until the second phase. This allows the corresponding parliamentary protocol to execute the first phase just once for all decrees, so the exchange of only three messages is needed to pass each individual decree.

The theorems on which the Synod protocol is based are similar to results obtained

[14] These algorithms were derived from the military protocols of another Mediterranean state.

by Dwork, Lynch, and Stockmeyer [Dwork et al. 1988]. However, their algorithms execute ballots sequentially in separate rounds, and they seem to be unrelated to the Synod protocol.

Much research has been done in the field since this article was written. The state-machine approach has been surveyed by Schneider [1990]. The recovery protocol by Keidar and Dolev [1996] and the totally-ordered broadcast algorithm of Fekete et al. [1997] are quite similar to the Paxon protocol described here. The author was also apparently unaware that the view management protocol by Oki and Liskov [1988] seems to be equivalent to the Paxon protocol.

Many of the refinements presented in this submission have also appeared in contemporary or subsequent articles. The method of delegation described in Section 3.3.3 is very similar to the leases mechanism of Gray and Cheriton [1989]. The technique of Section 3.3.4 in which the Paxons satisfy the monotonicity condition by using decree numbers is described by Ladin et al. [1992]. The technique of Section 3.3.6 for adding new legislators was also given by Schneider [1990].

K. M.

Appendix: Proof of Consistency of the Synodic Protocol

A1 The Basic Protocol

The Synod's basic protocol, described informally in Section 2.3, is stated here using modern algorithmic notation. We begin with the variables that a priest p must maintain. First come the variables that represent information kept in his ledger. (For convenience, the vote $prevVote[p]$ used in Section 2.3 is replaced by its components $prevBal[p]$ and $prevDec[p]$.)

- $outcome[p]$ — The decree written in p's ledger, or BLANK if there is nothing written there yet.
- $lastTried[p]$ — The number of the last ballot that p tried to begin, or $-\infty$ if there was none.
- $prevBal[p]$ — The number of the last ballot in which p voted, or $-\infty$ if he never voted.
- $prevDec[p]$ — The decree for which p last voted, or BLANK if p never voted.
- $nextBal[p]$ — The number of the last ballot in which p agreed to participate, or $-\infty$ if he has never agreed to participate in a ballot.

Next come variables representing information that priest p could keep on a slip of paper:

- $status[p]$ — One of the following values:
  - *idle* — Not conducting or trying to begin a ballot
  - *trying* — Trying to begin ballot number $lastTried[p]$
  - *polling* — Now conducting ballot number $lastTried[p]$

  If p has lost his slip of paper, then $status[p]$ is assumed to equal *idle* and the values of the following four variables are irrelevant.
- $prevVotes[p]$ — The set of votes received in *LastVote* messages for the current ballot (the one with ballot number $lastTried[p]$).

quorum[*p*]	If *status*[*p*] = *polling*, then the set of priests forming the quorum of the current ballot; otherwise, meaningless.
voters[*p*]	If *status*[*p*] = *polling*, then the set of quorum members from whom *p* has received *Voted* messages in the current ballot; otherwise, meaningless.
decree[*p*]	If *status*[*p*] = *polling*, then the decree of the current ballot; otherwise, meaningless.

There is also the history variable $\mathcal{B}$, which is the set of ballots that have been started and their progress—namely, which priests have cast votes. (A history variable is one used in the development and proof of an algorithm, but not actually implemented.)

Next come the actions that priest *p* may take. These actions are assumed to be atomic, meaning that once an action is begun, it must be completed before priest *p* begins any other action. An action is described by an enabling condition and a list of effects. The enabling condition describes when the action can be performed; actions that receive a message are enabled whenever a messenger has arrived with the appropriate message. The list of effects describes how the action changes the algorithm's variables and what message, if any, it sends. (Each individual action sends at most one message.)

Recall that ballot numbers were partitioned among the priests. For any ballot number *b*, the Paxons defined *owner*(*b*) to be the priest who was allowed to use that ballot number.

The actions in the basic protocol are allowed actions; the protocol does not require that a priest ever do anything. No attempt at efficiency has been made; the actions allow *p* to do silly things, such as sending another *BeginBallot* message to a priest from whom he has already received a *LastVote* message.

Try New Ballot

Always enabled.

- Set *lastTried*[*p*] to any ballot number *b*, greater than its previous value, such that *owner*(*b*) = *p*.
- Set *status*[*p*] to *trying*.
- Set *prevVotes*[*p*] to $\emptyset$.

Send *NextBallot* **Message**

Enabled whenever *status*[*p*]=*trying*.

- Send a *NextBallot*(*lastTried*[*p*]) message to any priest.

Receive *NextBallot*(*b*) **Message**

If $b \geq$ *nextBal*[*p*] then

- Set *nextBal*[*p*] to *b*.

Send *LastVote* **Message**

Enabled whenever *nextBal*[*p*] > *prevBal*[*p*].

- Send a *LastVote*(*nextBal*[*p*], *v*) message to priest *owner*(*nextBal*[*p*]), where $v_{pst} = p$, $v_{bal} =$ *prevBal*[*p*], and $v_{dec} =$ *prevDec*[*p*].

Receive *LastVote*(b, v) **Message**

If $b = \mathit{lastTried}[p]$ and $\mathit{status}[p] = \mathit{trying}$, then

- Set $\mathit{prevVotes}[p]$ to the union of its original value and $\{v\}$.

Start Polling Majority Set Q

Enabled when $\mathit{status}[p] = \mathit{trying}$ and $Q \subseteq \{v_{pst} : v \in \mathit{prevVotes}[p]\}$, where Q is a majority set.

- Set $\mathit{status}[p]$ to *polling*.
- Set $\mathit{quorum}[p]$ to Q.
- Set $\mathit{voters}[p]$ to $\emptyset$.
- Set $\mathit{decree}[p]$ to a decree d chosen as follows: Let v be the maximum element of $\mathit{prevVotes}[p]$. If $v_{bal} \neq -\infty$ then $d = v_{dec}$, else d can equal any decree.
- Set $\mathcal{B}$ to the union of its former value and $\{B\}$, where $B_{dec} = d$, $B_{qrm} = Q$, $B_{vot} = \emptyset$, and $B_{bal} = \mathit{lastTried}[p]$.

Send *BeginBallot* **Message**

Enabled when $\mathit{status}[p] = \mathit{polling}$.

- Send a *BeginBallot*$(\mathit{lastTried}[p],\ \mathit{decree}[p])$ message to any priest in $\mathit{quorum}[p]$.

Receive *BeginBallot*$(b,\ d)$ **Message**

If $b = \mathit{nextBal}[p] > \mathit{prevBal}[p]$ then

- Set $\mathit{prevBal}[p]$ to b.
- Set $\mathit{prevDec}[p]$ to d.
- If there is a ballot B in $\mathcal{B}$ with $B_{bal} = b$ [there will be], then choose any such B [there will be only one] and let the new value of $\mathcal{B}$ be obtained from its old value by setting B_{vot} equal to the union of its old value and $\{p\}$.

Send *Voted* **Message**

Enabled whenever $\mathit{prevBal}[p] \neq -\infty$.

- Send a *Voted*$(\mathit{prevBal}[p],\ p)$ message to $\mathit{owner}(\mathit{prevBal}[p])$.

Receive *Voted*$(b,\ q)$ **Message**

If $b = \mathit{lastTried}[p]$ and $\mathit{status}[p] = \mathit{polling}$, then

- Set $\mathit{voters}[p]$ to the union of its old value and $\{q\}$

Succeed

Enabled whenever $\mathit{status}[p] = \mathit{polling}$, $\mathit{quorum}[p] \subseteq \mathit{voters}[p]$, and $\mathit{outcome}[p] =$ BLANK.

- Set $\mathit{outcome}[p]$ to $\mathit{decree}[p]$.

Send *Success* **Message**

Enabled whenever $\mathit{outcome}[p] \neq$ BLANK.

- Send a *Success*$(\mathit{outcome}[p])$ message to any priest.

Receive *Success*(d) **Message**

If $\mathit{outcome}[p] =$ BLANK, then

– Set $outcome[p]$ to d.

This algorithm is an abstract description of the real protocol performed by Paxon priests. Do the algorithm's actions accurately model the actions of the real priests? There were three kinds of actions that a priest could perform "atomically": receiving a message, writing a note or ledger entry, and sending a message. Each of these is represented by a single action of the algorithm, except that **Receive** actions both receive a message and set a variable. We can pretend that the receipt of a message occurred when a priest acted upon the message; if he left the Chamber before acting upon it, then we can pretend that the message was never received. Since this pretense does not affect the consistency condition, we can infer the consistency of the basic Synod protocol from the consistency of the algorithm.

A2 Proof of Consistency

To prove the consistency condition, it is necessary to show that whenever $outcome[p]$ and $outcome[q]$ are both different from BLANK, they are equal. A rigorous correctness proof requires a complete description of the algorithm. The description given above is almost complete. Missing is a variable $\mathcal{M}$ whose value is the multiset of all messages in transit.[15] Each **Send** action adds a message to this multiset and each **Receive** action removes one. Also needed are actions to represent the loss and duplication of messages, as well as a **Forget** action that represents a priest losing his slip of paper.

With these additions, we get an algorithm that defines a set of possible behaviors, in which each change of state corresponds to one of the allowed actions. The Paxons proved correctness by finding a predicate I such that

(1) I is true initially.

(2) I implies the desired correctness condition.

(3) Each allowed action leaves I true.

The predicate I was written as a conjunction $I1 \wedge \ldots \wedge I7$, where $I1$–$I5$ were in turn the conjunction of predicates $I1(p)$–$I5(p)$ for all priests p. Although most variables are mentioned in several of the conjuncts, each variable except $status[p]$ is naturally associated with one conjunct, and each conjunct can be thought of as a constraint on its associated variables. The definitions of the individual conjuncts of I are given below, where a list of items marked by $\wedge$ symbols denotes the conjunction of those items. The variables associated with a conjunct are listed in bracketed comments.

$I1(p) \triangleq$ [Associated variable: $outcome[p]$]

$(outcome[p] \neq \text{BLANK}) \Rightarrow \exists B \in \mathcal{B} : (B_{qrm} \subseteq B_{vot}) \wedge (B_{dec} = outcome[p])$

$I2(p) \triangleq$ [Associated variable: $lastTried[p]$]

$\wedge\ owner(lastTried[p]) = p$
$\wedge\ \forall B \in \mathcal{B} : (owner(B_{bal}) = p) \Rightarrow$
$\qquad \wedge\ B_{bal} \leq lastTried[p]$
$\qquad \wedge\ (status[p] = trying) \Rightarrow (B_{bal} < lastTried[p])$

[15] A multiset is a set that may contain multiple copies of the same element.

$I3(p) \triangleq$ [Associated variables: $prevBal[p]$, $prevDec[p]$, $nextBal[p]$]

$\land\ prevBal[p] = MaxVote(\infty, p, \mathcal{B})_{bal}$
$\land\ prevDec[p] = MaxVote(\infty, p, \mathcal{B})_{dec}$
$\land\ nextBal[p] \geq prevBal[p]$

$I4(p) \triangleq$ [Associated variable: $prevVotes[p]$]

$(status[p] \neq idle) \Rightarrow$
$\quad \forall v \in prevVotes[p] : \land\ v = MaxVote(lastTried[p], v_{pst}, \mathcal{B})$
$\qquad\qquad\qquad\qquad \land\ nextBal[v_{pst}] \geq lastTried[p]$

$I5(p) \triangleq$ [Associated variables: $quorum[p]$, $voters[p]$, $decree[p]$]

$(status[p] = polling) \Rightarrow$
$\quad \land\ quorum[p] \subseteq \{v_{pst} : v \in prevVotes[p]\}$
$\quad \land\ \exists B \in \mathcal{B} : \land\ quorum[p] = B_{qrm}$
$\qquad\qquad\qquad \land\ decree[p] = B_{dec}$
$\qquad\qquad\qquad \land\ voters[p] \subseteq B_{vot}$
$\qquad\qquad\qquad \land\ lastTried[p] = B_{bal}$

$I6 \triangleq$ [Associated variable: $\mathcal{B}$]

$\land\ B1(\mathcal{B}) \land B2(\mathcal{B}) \land B3(\mathcal{B})$
$\land\ \forall B \in \mathcal{B} :\ B_{qrm}$ is a majority set

$I7 \triangleq$ [Associated variable: $\mathcal{M}$]

$\land\ \forall NextBallot(b) \in \mathcal{M} : (b \leq lastTried[owner(b)])$
$\land\ \forall LastVote(b, v) \in \mathcal{M} : \land\ v = MaxVote(b, v_{pst}, \mathcal{B})$
$\qquad\qquad\qquad\qquad\qquad\qquad \land\ nextBal[v_{pst}] \geq b$
$\land\ \forall BeginBallot(b, d) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \land (B_{dec} = d)$
$\land\ \forall Voted(b, p) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \land (p \in B_{vot})$
$\land\ \forall Success(d) \in \mathcal{M} : \exists p : outcome[p] = d \neq \text{BLANK}$

The Paxons had to prove that I satisfies the three conditions given above. The first condition, that I holds initially, requires checking that each conjunct is true for the initial values of all the variables. While not stated explicitly, these initial values can be inferred from the variables' descriptions, and checking the first condition is straightforward. The second condition, that I implies consistency, follows from $I1$, the first conjunct of $I6$, and Theorem 1. The hard part was proving the third condition, the invariance of I, which meant proving that I is left true by every action. This condition is proved by showing that, for each conjunct of I, executing any action when I is true leaves that conjunct true. The proofs are sketched below.

$I1(p)$ $\mathcal{B}$ is changed only by adding a new ballot or adding a new priest to B_{vot} for some $B \in \mathcal{B}$, neither of which can falsify $I1(p)$. The value of $outcome[p]$ is changed only by the **Succeed** and **Receive** *Success* **Message** actions. The enabling condition and $I5(p)$ imply that $I1(p)$ is left true by the **Succeed** action. The enabling condition, $I1(p)$, and the last conjunct of $I7$ imply that $I1(p)$ is left true by the **Receive** *Success* **Message** action.

$I2(p)$ This conjunct depends only on $lastTried[p]$, $status[p]$, and $\mathcal{B}$. Only the **Try New Ballot** action changes $lastTried[p]$, and only that action can set $status[p]$ to *trying*. Since the action increases $lastTried[p]$ to a value b with $owner(b) = p$, it leaves $I2(p)$ true. A completely new element is added to $\mathcal{B}$ only by a **Start Polling** action; the first conjunct of $I2(p)$ and the specification of the action imply that adding this new element does not falsify the second conjunct of $I2(p)$. The only other way $\mathcal{B}$ is changed is by adding a new priest to B_{vot} for some $B \in \mathcal{B}$, which does not affect $I2(p)$.

$I3(p)$ Since votes are never removed from $\mathcal{B}$, the only action that can change $MaxVote(\infty, p, \mathcal{B})$ is one that adds to $\mathcal{B}$ a vote cast by p. Only a **Receive** *BeginBallot* **Message** action can do that, and only that action changes $prevBal[p]$ and $prevDec[p]$. The *BeginBallot* conjunct of $I7$ implies that this action actually does add a vote to $\mathcal{B}$, and $B1(\mathcal{B})$ (the first conjunct of $I6$) implies that there is only one ballot to which the vote can be added. The enabling condition, the assumption that $I3(p)$ holds before executing the action, and the definition of *MaxVote* then imply that the action leaves the first two conjuncts of $I3(p)$ true. The third conjunct is left true because $prevBal[p]$ is changed only by setting it to $nextBal[p]$, and $nextBal[p]$ is never decreased.

$I4(p)$ This conjunct depends only upon the values of $status[p]$, $prevVotes[p]$, $lastTried[p]$, $nextBal[q]$ for some priests q, and $\mathcal{B}$. The value of $status[p]$ is changed from *idle* to not *idle* only by a **Try New Ballot** action, which sets $prevVotes[p]$ to $\emptyset$, making $I4(p)$ vacuously true. The only other actions that change $prevVotes[p]$ are the **Forget** action, which leaves $I4(p)$ true because it sets $status[p]$ to *idle*, and the **Receive** *LastVote* **Message** action. It follows from the enabling condition and the *LastVote* conjunct of $I7$ that the **Receive** *LastVote* **Message** action preserves $I4(p)$. The value of $lastTried[p]$ is changed only by the **Try New Ballot** action, which leaves $I4(p)$ true because it sets $status[p]$ to *trying*. The value of $nextBal[q]$ can only increase, which cannot make $I4(p)$ false. Finally, $MaxVote(lastTried[p], v_{pst}, \mathcal{B})$ can be changed only if v_{pst} is added to B_{vot} for some $B \in \mathcal{B}$ with $B_{bal} < lastTried[p]$. But v_{pst} is added to B_{vot} (by a **Receive** *BeginBallot* **Message** action) only if $nextBal[v_{pst}] = B_{bal}$, in which case $I4(p)$ implies that $B_{bal} \geq lastTried[p]$.

$I5(p)$ The value of $status[p]$ is set to *polling* only by the **Start Polling** action. This action's enabling condition guarantees that the first conjunct becomes true, and it adds the ballot to $\mathcal{B}$ that makes the second conjunct true. No other action changes $quorum[p]$, $decree[p]$, or $lastTried[p]$ while leaving $status[p]$ equal to *polling*. The value of $prevVotes[p]$ cannot be changed while $status[p] = polling$, and $\mathcal{B}$ is changed only by adding new elements or by adding a new priest to B_{vot}. The only remaining possibility for falsifying $I5(p)$ is the addition of a new element to $voters[p]$ by the **Receive** *Voted* **Message** action. The *Voted* conjunct of $I7$, $B1(\mathcal{B})$ (the first conjunct of $I6$), and the action's enabling condition imply that the element added to $voters[p]$ is in B_{vot}, where B is the ballot whose existence is asserted in $I5(p)$.

$I6$ Since B_{bal} and B_{qrm} are never changed for any $B \in \mathcal{B}$, the only way $B1(\mathcal{B})$, $B2(\mathcal{B})$, and the second conjunct of $I6$ can be falsified is by adding a new ballot to

$\mathcal{B}$, which is done only by the **Start Polling Majority Set** Q action when *status*[p] equals *trying*. It follows from the second conjunct of $I2(p)$ that this action leaves $B1(\mathcal{B})$ true; and the assertion, in the enabling condition, that Q is a majority set implies that the action leaves $B2(\mathcal{B})$ and the second conjunct of $I6$ true. There are two possible ways of falsifying $B3(\mathcal{B})$: changing $MaxVote(B_{bal}, B_{qrm}, \mathcal{B})$ by adding a new vote to $\mathcal{B}$, and adding a new ballot to $\mathcal{B}$. A new vote is added only by the **Receive** *BeginBallot* **Message** action, and $I3(p)$ implies that the action adds a vote later than any other vote cast by p in $\mathcal{B}$, so it cannot change $MaxVote(B_{bal}, B_{qrm}, \mathcal{B})$ for any B in $\mathcal{B}$. Conjunct $I4(p)$ implies that the new ballot added by the **Start Polling** action does not falsify $B3(\mathcal{B})$.

I7 $I7$ can be falsified either by adding a new message to $\mathcal{M}$ or by changing the value of another variable on which $I7$ depends. Since *lastTried*[p] and *nextBal*[p] are never decreased, changing them cannot make $I7$ false. Since *outcome*[p] is never changed if its value is not BLANK, changing it cannot falsify $I7$. Since $\mathcal{B}$ is changed only by adding ballots and adding votes, the only change to it that can make $I7$ false is the addition of a vote by v_{pst} that makes the $LastVote(b, v)$ conjunct false by changing $MaxVote(b, v_{pst}, \mathcal{B})$. This can happen only if v_{pst} votes in a ballot B with $B_{bal} < b$. But v_{pst} can vote only in ballot number $nextBal[v_{pst}]$, and the assumption that this conjunct holds initially implies that $nextBal[v_{pst}] \geq b$. Therefore, we need check only that every message that is sent satisfies the condition in the appropriate conjunct of $I7$.

NextBallot: Follows from the definition of the **Send** *NextBallot* **Message** action and the first conjunct of $I2(p)$.

LastVote: The enabling condition of the **Send** *LastVote* **Message** action and $I3(p)$ imply that $MaxVote(nextBal[p], p, \mathcal{B}) = MaxVote(\infty, p, \mathcal{B})$, from which it follows that the *LastVote* message sent by the action satisfies the condition in $I7$.

BeginBallot: Follows from $I5(p)$ and the definition of the **Send** *BeginBallot* **Message** action.

Voted: Follows from $I3(p)$, the definition of *MaxVote*, and the definition of the **Send** *Voted* **Message** action.

Success: Follows from the definition of **Send** *Success* **Message**.

ACKNOWLEDGMENTS

Daniel Duchamp pointed out to me the need for a new state-machine implementation. Discussions with Martín Abadi, Andy Hisgen, Tim Mann, and Garret Swart led me to Paxos. *Λεωνίδας Γκίμπας* provided invaluable assistance with the Paxon dialect.

References

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts.

De Prisco, R., Lampson, B., and Lynch, N. 1997. Revisiting the Paxos algorithm. In M. Mavronicolas and P. Tsigas (Eds.), *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG 97)*, Volume 1320 of *Lecture Notes in Computer*

Science, Saarbruken, Germany, pp. 111–125. Springer-Verlag.

Dijkstra, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM 17,* 11 (Nov.), 643–644.

Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35,* 2 (April), 288–323.

Fekete, A., Lynch, N., and Shvartsman, A. 1997. Specifying and using a partitionable group communication service. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 53–62. ACM Press.

Fischer, M. J., Lynch, N., and Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32,* 2 (April), 374–382.

Gray, C. G. and Cheriton, D. R. 1989. Leases: An efficient fault-toerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, New York, pp. 202–210. ACM.

Keidar, I. and Dolev, D. 1996. Efficient message ordering in dynamic networks. In *Proceedings of the* 15th *Annual ACM Symposium on Principles of Distributed Computing*. ACM.

Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems 10,* 4 (Nov.), 360–391.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (July), 558–565.

Lamport, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. on Programm. Lang. Syst. 6,* 2 (April), 254–280.

Lampson, B. W. 1996. How to build a highly available system using consensus. In O. Babaoglu and K. Marzullo (Eds.), *Distributed Algorithms*, Volume 1151 of *Lecture Notes in Computer Science*, Berlin, pp. 1–17. Springer-Verlag.

Oki, B. M. and Liskov, B. H. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pp. 8–17. ACM Press.

Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22,* 4 (Dec.), 299–319.

Skeen, M. D. 1982. Crash recovery in a distributed database system. Ph. D. thesis, University of California, Berkeley.

Received January 1990; Accepted March 1998

Chapter 5

Paxos Made Simple

Authors:
Leslie Lamport

Publication year: 2001

Paxos Made Simple

Leslie Lamport

01 Nov 2001

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

Contents

1 Introduction **1**

2 The Consensus Algorithm **1**

2.1 The Problem 1

2.2 Choosing a Value 2

2.3 Learning a Chosen Value 6

2.4 Progress 7

2.5 The Implementation 7

3 Implementing a State Machine **8**

References **11**

1 Introduction

The Paxos algorithm for implementing a fault-tolerant distributed system has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers [5]. In fact, it is among the simplest and most obvious of distributed algorithms. At its heart is a consensus algorithm—the "synod" algorithm of [5]. The next section shows that this consensus algorithm follows almost unavoidably from the properties we want it to satisfy. The last section explains the complete Paxos algorithm, which is obtained by the straightforward application of consensus to the state machine approach for building a distributed system—an approach that should be well-known, since it is the subject of what is probably the most often-cited article on the theory of distributed systems [4].

2 The Consensus Algorithm

2.1 The Problem

Assume a collection of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen. If no value is proposed, then no value should be chosen. If a value has been chosen, then processes should be able to learn the chosen value. The safety requirements for consensus are:

- Only a value that has been proposed may be chosen,
- Only a single value is chosen, and
- A process never learns that a value has been chosen unless it actually has been.

We won't try to specify precise liveness requirements. However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

We let the three roles in the consensus algorithm be performed by three classes of agents: *proposers*, *acceptors*, and *learners*. In an implementation, a single process may act as more than one agent, but the mapping from agents to processes does not concern us here.

Assume that agents can communicate with one another by sending messages. We use the customary asynchronous, non-Byzantine model, in which:

- Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.

- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.

2.2 Choosing a Value

The easiest way to choose a value is to have a single acceptor agent. A proposer sends a proposal to the acceptor, who chooses the first proposed value that it receives. Although simple, this solution is unsatisfactory because the failure of the acceptor makes any further progress impossible.

So, let's try another way of choosing a value. Instead of a single acceptor, let's use multiple acceptor agents. A proposer sends a proposed value to a set of acceptors. An acceptor may *accept* the proposed value. The value is chosen when a large enough set of acceptors have accepted it. How large is large enough? To ensure that only a single value is chosen, we can let a large enough set consist of any majority of the agents. Because any two majorities have at least one acceptor in common, this works if an acceptor can accept at most one value. (There is an obvious generalization of a majority that has been observed in numerous papers, apparently starting with [3].)

In the absence of failure or message loss, we want a value to be chosen even if only one value is proposed by a single proposer. This suggests the requirement:

P1. An acceptor must accept the first proposal that it receives.

But this requirement raises a problem. Several values could be proposed by different proposers at about the same time, leading to a situation in which every acceptor has accepted a value, but no single value is accepted by a majority of them. Even with just two proposed values, if each is accepted by about half the acceptors, failure of a single acceptor could make it impossible to learn which of the values was chosen.

P1 and the requirement that a value is chosen only when it is accepted by a majority of acceptors imply that an acceptor must be allowed to accept more than one proposal. We keep track of the different proposals that an acceptor may accept by assigning a (natural) number to each proposal, so a proposal consists of a proposal number and a value. To prevent confusion, we require that different proposals have different numbers. How this is

achieved depends on the implementation, so for now we just assume it. A value is chosen when a single proposal with that value has been accepted by a majority of the acceptors. In that case, we say that the proposal (as well as its value) has been chosen.

We can allow multiple proposals to be chosen, but we must guarantee that all chosen proposals have the same value. By induction on the proposal number, it suffices to guarantee:

P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.

Since numbers are totally ordered, condition P2 guarantees the crucial safety property that only a single value is chosen.

To be chosen, a proposal must be accepted by at least one acceptor. So, we can satisfy P2 by satisfying:

P2^a. If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v.

We still maintain P1 to ensure that some proposal is chosen. Because communication is asynchronous, a proposal could be chosen with some particular acceptor c never having received any proposal. Suppose a new proposer "wakes up" and issues a higher-numbered proposal with a different value. P1 requires c to accept this proposal, violating P2^a. Maintaining both P1 and P2^a requires strengthening P2^a to:

P2^b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.

Since a proposal must be issued by a proposer before it can be accepted by an acceptor, P2^b implies P2^a, which in turn implies $P2$.

To discover how to satisfy P2^b, let's consider how we would prove that it holds. We would assume that some proposal with number m and value v is chosen and show that any proposal issued with number $n > m$ also has value v. We would make the proof easier by using induction on n, so we can prove that proposal number n has value v under the additional assumption that every proposal issued with a number in $m \mathinner{..} (n-1)$ has value v, where $i \mathinner{..} j$ denotes the set of numbers from i through j. For the proposal numbered m to be chosen, there must be some set C consisting of a majority of acceptors such that every acceptor in C accepted it. Combining this with the induction assumption, the hypothesis that m is chosen implies:

Every acceptor in C has accepted a proposal with number in $m\,..\,(n-1)$, and every proposal with number in $m\,..\,(n-1)$ accepted by any acceptor has value v.

Since any set S consisting of a majority of acceptors contains at least one member of C, we can conclude that a proposal numbered n has value v by ensuring that the following invariant is maintained:

P2^c. For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.

We can therefore satisfy P2^b by maintaining the invariance of P2^c.

To maintain the invariance of P2^c, a proposer that wants to issue a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors. Learning about proposals already accepted is easy enough; predicting future acceptances is hard. Instead of trying to predict the future, the proposer controls it by extracting a promise that there won't be any such acceptances. In other words, the proposer requests that the acceptors not accept any more proposals numbered less than n. This leads to the following algorithm for issuing proposals.

1. A proposer chooses a new proposal number n and sends a request to each member of some set of acceptors, asking it to respond with:

   (a) A promise never again to accept a proposal numbered less than n, and

   (b) The proposal with the highest number less than n that it has accepted, if any.

   I will call such a request a *prepare* request with number n.

2. If the proposer receives the requested responses from a majority of the acceptors, then it can issue a proposal with number n and value v, where v is the value of the highest-numbered proposal among the responses, or is any value selected by the proposer if the responders reported no proposals.

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted. (This need not be the same set of acceptors that responded to the initial requests.) Let's call this an *accept* request.

This describes a proposer's algorithm. What about an acceptor? It can receive two kinds of requests from proposers: *prepare* requests and *accept* requests. An acceptor can ignore any request without compromising safety. So, we need to say only when it is allowed to respond to a request. It can always respond to a *prepare* request. It can respond to an *accept* request, accepting the proposal, iff it has not promised not to. In other words:

P1^a. An acceptor can accept a proposal numbered n iff it has not responded to a *prepare* request having a number greater than n.

Observe that P1^a subsumes P1.

We now have a complete algorithm for choosing a value that satisfies the required safety properties—assuming unique proposal numbers. The final algorithm is obtained by making one small optimization.

Suppose an acceptor receives a *prepare* request numbered n, but it has already responded to a *prepare* request numbered greater than n, thereby promising not to accept any new proposal numbered n. There is then no reason for the acceptor to respond to the new *prepare* request, since it will not accept the proposal numbered n that the proposer wants to issue. So we have the acceptor ignore such a *prepare* request. We also have it ignore a *prepare* request for a proposal it has already accepted.

With this optimization, an acceptor needs to remember only the highest-numbered proposal that it has ever accepted and the number of the highest-numbered *prepare* request to which it has responded. Because P2^c must be kept invariant regardless of failures, an acceptor must remember this information even if it fails and then restarts. Note that the proposer can always abandon a proposal and forget all about it—as long as it never tries to issue another proposal with the same number.

Putting the actions of the proposer and acceptor together, we see that the algorithm operates in the following two phases.

Phase 1. (a) A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors.

(b) If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2. (a) If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.

A proposer can make multiple proposals, so long as it follows the algorithm for each one. It can abandon a proposal in the middle of the protocol at any time. (Correctness is maintained, even though requests and/or responses for the proposal may arrive at their destinations long after the proposal was abandoned.) It is probably a good idea to abandon a proposal if some proposer has begun trying to issue a higher-numbered one. Therefore, if an acceptor ignores a *prepare* or *accept* request because it has already received a *prepare* request with a higher number, then it should probably inform the proposer, who should then abandon its proposal. This is a performance optimization that does not affect correctness.

2.3 Learning a Chosen Value

To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors. The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal. This allows learners to find out about a chosen value as soon as possible, but it requires each acceptor to respond to each learner—a number of responses equal to the product of the number of acceptors and the number of learners.

The assumption of non-Byzantine failures makes it easy for one learner to find out from another learner that a value has been accepted. We can have the acceptors respond with their acceptances to a distinguished learner, which in turn informs the other learners when a value has been chosen. This approach requires an extra round for all the learners to discover the chosen value. It is also less reliable, since the distinguished learner could fail. But it requires a number of responses equal only to the sum of the number of acceptors and the number of learners.

More generally, the acceptors could respond with their acceptances to some set of distinguished learners, each of which can then inform all the learners when a value has been chosen. Using a larger set of distinguished

learners provides greater reliability at the cost of greater communication complexity.

Because of message loss, a value could be chosen with no learner ever finding out. The learner could ask the acceptors what proposals they have accepted, but failure of an acceptor could make it impossible to know whether or not a majority had accepted a particular proposal. In that case, learners will find out what value is chosen only when a new proposal is chosen. If a learner needs to know whether a value has been chosen, it can have a proposer issue a proposal, using the algorithm described above.

2.4 Progress

It's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen. Proposer p completes phase 1 for a proposal number n_1. Another proposer q then completes phase 1 for a proposal number $n_2 > n_1$. Proposer p's phase 2 *accept* requests for a proposal numbered n_1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n_2. So, proposer p then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 *accept* requests of proposer q to be ignored. And so on.

To guarantee progress, a distinguished proposer must be selected as the only one to try issuing proposals. If the distinguished proposer can communicate successfully with a majority of acceptors, and if it uses a proposal with number greater than any already used, then it will succeed in issuing a proposal that is accepted. By abandoning a proposal and trying again if it learns about some request with a higher proposal number, the distinguished proposer will eventually choose a high enough proposal number.

If enough of the system (proposer, acceptors, and communication network) is working properly, liveness can therefore be achieved by electing a single distinguished proposer. The famous result of Fischer, Lynch, and Patterson [1] implies that a reliable algorithm for electing a proposer must use either randomness or real time—for example, by using timeouts. However, safety is ensured regardless of the success or failure of the election.

2.5 The Implementation

The Paxos algorithm [5] assumes a network of processes. In its consensus algorithm, each process plays the role of proposer, acceptor, and learner. The algorithm chooses a leader, which plays the roles of the distinguished

proposer and the distinguished learner. The Paxos consensus algorithm is precisely the one described above, where requests and responses are sent as ordinary messages. (Response messages are tagged with the corresponding proposal number to prevent confusion.) Stable storage, preserved during failures, is used to maintain the information that the acceptor must remember. An acceptor records its intended response in stable storage before actually sending the response.

All that remains is to describe the mechanism for guaranteeing that no two proposals are ever issued with the same number. Different proposers choose their numbers from disjoint sets of numbers, so two different proposers never issue a proposal with the same number. Each proposer remembers (in stable storage) the highest-numbered proposal it has tried to issue, and begins phase 1 with a higher proposal number than any it has already used.

3 Implementing a State Machine

A simple way to implement a distributed system is as a collection of clients that issue commands to a central server. The server can be described as a deterministic state machine that performs client commands in some sequence. The state machine has a current state; it performs a step by taking as input a command and producing an output and a new state. For example, the clients of a distributed banking system might be tellers, and the state-machine state might consist of the account balances of all users. A withdrawal would be performed by executing a state machine command that decreases an account's balance if and only if the balance is greater than the amount withdrawn, producing as output the old and new balances.

An implementation that uses a single central server fails if that server fails. We therefore instead use a collection of servers, each one independently implementing the state machine. Because the state machine is deterministic, all the servers will produce the same sequences of states and outputs if they all execute the same sequence of commands. A client issuing a command can then use the output generated for it by any server.

To guarantee that all servers execute the same sequence of state machine commands, we implement a sequence of separate instances of the Paxos consensus algorithm, the value chosen by the ith instance being the ith state machine command in the sequence. Each server plays all the roles (proposer, acceptor, and learner) in each instance of the algorithm. For now, I assume that the set of servers is fixed, so all instances of the consensus algorithm

use the same sets of agents.

In normal operation, a single server is elected to be the leader, which acts as the distinguished proposer (the only one that tries to issue proposals) in all instances of the consensus algorithm. Clients send commands to the leader, who decides where in the sequence each command should appear. If the leader decides that a certain client command should be the 135th command, it tries to have that command chosen as the value of the 135th instance of the consensus algorithm. It will usually succeed. It might fail because of failures, or because another server also believes itself to be the leader and has a different idea of what the 135th command should be. But the consensus algorithm ensures that at most one command can be chosen as the 135th one.

Key to the efficiency of this approach is that, in the Paxos consensus algorithm, the value to be proposed is not chosen until phase 2. Recall that, after completing phase 1 of the proposer's algorithm, either the value to be proposed is determined or else the proposer is free to propose any value.

I will now describe how the Paxos state machine implementation works during normal operation. Later, I will discuss what can go wrong. I consider what happens when the previous leader has just failed and a new leader has been selected. (System startup is a special case in which no commands have yet been proposed.)

The new leader, being a learner in all instances of the consensus algorithm, should know most of the commands that have already been chosen. Suppose it knows commands 1–134, 138, and 139—that is, the values chosen in instances 1–134, 138, and 139 of the consensus algorithm. (We will see later how such a gap in the command sequence could arise.) It then executes phase 1 of instances 135–137 and of all instances greater than 139. (I describe below how this is done.) Suppose that the outcome of these executions determine the value to be proposed in instances 135 and 140, but leaves the proposed value unconstrained in all other instances. The leader then executes phase 2 for instances 135 and 140, thereby choosing commands 135 and 140.

The leader, as well as any other server that learns all the commands the leader knows, can now execute commands 1–135. However, it can't execute commands 138–140, which it also knows, because commands 136 and 137 have yet to be chosen. The leader could take the next two commands requested by clients to be commands 136 and 137. Instead, we let it fill the gap immediately by proposing, as commands 136 and 137, a special "no-op" command that leaves the state unchanged. (It does this by executing phase 2 of instances 136 and 137 of the consensus algorithm.) Once these

no-op commands have been chosen, commands 138–140 can be executed.

Commands 1–140 have now been chosen. The leader has also completed phase 1 for all instances greater than 140 of the consensus algorithm, and it is free to propose any value in phase 2 of those instances. It assigns command number 141 to the next command requested by a client, proposing it as the value in phase 2 of instance 141 of the consensus algorithm. It proposes the next client command it receives as command 142, and so on.

The leader can propose command 142 before it learns that its proposed command 141 has been chosen. It's possible for all the messages it sent in proposing command 141 to be lost, and for command 142 to be chosen before any other server has learned what the leader proposed as command 141. When the leader fails to receive the expected response to its phase 2 messages in instance 141, it will retransmit those messages. If all goes well, its proposed command will be chosen. However, it could fail first, leaving a gap in the sequence of chosen commands. In general, suppose a leader can get α commands ahead—that is, it can propose commands $i+1$ through $i+\alpha$ after commands 1 through i are chosen. A gap of up to $\alpha-1$ commands could then arise.

A newly chosen leader executes phase 1 for infinitely many instances of the consensus algorithm—in the scenario above, for instances 135–137 and all instances greater than 139. Using the same proposal number for all instances, it can do this by sending a single reasonably short message to the other servers. In phase 1, an acceptor responds with more than a simple OK only if it has already received a phase 2 message from some proposer. (In the scenario, this was the case only for instances 135 and 140.) Thus, a server (acting as acceptor) can respond for all instances with a single reasonably short message. Executing these infinitely many instances of phase 1 therefore poses no problem.

Since failure of the leader and election of a new one should be rare events, the effective cost of executing a state machine command—that is, of achieving consensus on the command/value—is the cost of executing only phase 2 of the consensus algorithm. It can be shown that phase 2 of the Paxos consensus algorithm has the minimum possible cost of any algorithm for reaching agreement in the presence of faults [2]. Hence, the Paxos algorithm is essentially optimal.

This discussion of the normal operation of the system assumes that there is always a single leader, except for a brief period between the failure of the current leader and the election of a new one. In abnormal circumstances, the leader election might fail. If no server is acting as leader, then no new commands will be proposed. If multiple servers think they are leaders, then

they can all propose values in the same instance of the consensus algorithm, which could prevent any value from being chosen. However, safety is preserved—two different servers will never disagree on the value chosen as the ith state machine command. Election of a single leader is needed only to ensure progress.

If the set of servers can change, then there must be some way of determining what servers implement what instances of the consensus algorithm. The easiest way to do this is through the state machine itself. The current set of servers can be made part of the state and can be changed with ordinary state-machine commands. We can allow a leader to get α commands ahead by letting the set of servers that execute instance $i + \alpha$ of the consensus algorithm be specified by the state after execution of the ith state machine command. This permits a simple implementation of an arbitrarily sophisticated reconfiguration algorithm.

References

[1] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[2] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial. TechnicalReport MIT-LCS-TR-821, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, May 2001. also published in SIGACT News 32(2) (June 2001).

[3] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

Chapter 6

Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web

Authors:
Seth Gilbert, Nancy Lynch

Publication year: 2002

Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert* Nancy Lynch*

Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

1 Introduction

At PODC 2000, Brewer[1], in an invited talk [2], made the following conjecture: it is impossible for a web service to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

All three of these properties are desirable – and expected – from real-world web services. In this note, we will first discuss what Brewer meant by the conjecture; next we will formalize these concepts and prove the conjecture;

*Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

[1]Eric Brewer is a professor at the University of California, Berkeley, and the co-founder and Chief Scientist of Inktomi.

finally, we will describe and attempt to formalize some real-world solutions to this practical difficulty.

Most web services today attempt to provide strongly consistent data. There has been significant research designing ACID[2] databases, and most of the new frameworks for building distributed web services depend on these databases. Interactions with web services are expected to behave in a transactional manner: operations commit or fail in their entirety (atomic), committed transactions are visible to all future transactions (consistent), uncommitted transactions are isolated from each other (isolated), and once a transaction is committed it is permanent (durable). It is clearly important, for example, that billing information and commercial transaction records be handled with this type of strong consistency.

Web services are similarly expected to be highly available. Every request should succeed and receive a response. When a service goes down, it may well create significant real-world problems; the classic example of this is the potential legal difficulties should the E-Trade web site go down. This problem is exacerbated by the fact that a web-site is most likely to be unavailable when it is most needed. The goal of most web services today is to be as available as the network on which they run: if any service on the network is available, then the web service should be accessible.

Finally, on a highly distributed network, it is desirable to provide some amount of fault-tolerance. When some nodes crash or some communication links fail, it is important that the service still perform as expected. One desirable fault tolerance property is the ability to survive a network partitioning into multiple components. In this note we will not consider stopping failures, though in some cases a stopping failure can be modeled as a node existing in its own unique component of a partition.

2 Formal Model

In this section, we will formally define what is meant by the terms *consistent*, *available*, and *partition tolerant*.

2.1 Atomic Data Objects

The most natural way of formalizing the idea of a consistent service is as an atomic data object. Atomic [5], or linearizable [4], consistency is the

[2] **A**tomic, **C**onsistent, **I**solated, **D**urable

condition expected by most web services today.[3] Under this consistency guarantee, there must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time. One important property of an atomic read/write shared memory is that any read operation that begins after a write operation completes must return that value, or the result of a later write operation. This is the consistency guarantee that generally provides the easiest model for users to understand, and is most convenient for those attempting to design a client application that uses the distributed service. See [6] for a more complete definition of atomic consistency.

2.2 Available Data Objects

For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response.[4] That is, any algorithm used by the service must eventually terminate. In some ways this is a weak definition of availability: it puts no bound on how long the algorithm may run before terminating, and therefore allows unbounded computation. On the other hand, when qualified by the need for partition tolerance, this can be seen as a strong definition of availability: even when severe network failures occur, every request must terminate.

2.3 Partition Tolerance

The above definitions of availability and atomicity are qualified by the need to tolerate partitions. In order to model partition tolerance, the network will be allowed to lose arbitrarily many messages sent from one node to another. When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost. (And any pattern of message loss can be modeled as a temporary partition separating the communicating nodes at the exact instant the message is lost.)

[3] Discussing atomic consistency is somewhat different than talking about an ACID database, as database consistency refers to transactions, while atomic consistency refers only to a property of a single request/response operation sequence. And it has a different meaning than the **A**tomic in ACID, as it subsumes the database notions of both **A**tomic and **C**onsistent.

[4] Brewer originally only required *almost all* requests to receive a response. As allowing probabilistic availability does not change the result when arbitrary failures occur, for simplicity we are requiring 100% availability.

The atomicity requirement (§2.1) therefore implies that every response will be atomic, even though arbitrary messages sent as part of the algorithm might not be delivered. The availability requirement (§2.2) implies that every node receiving a request from a client must respond, even though arbitrary messages that are sent may be lost. Note that this is similar to wait-free termination in a pure shared-memory system: even if every other node in the network fails (i.e. the node is in its own unique component of the partition), a valid (atomic) response must be generated. No set of failures less than total network failure is allowed to cause the system to respond incorrectly.[5]

3 Asynchronous Networks

3.1 Impossibility Result

In proving this conjecture, we will use the asynchronous network model, as formalized by Lynch in [7]. In the asynchronous model, there is no clock, and nodes must make decisions based only on the messages received and local computation.

Theorem 1 *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability*
- *Atomic consistency*

in all fair executions (including those in which messages are lost).

Proof: We prove this by contradiction. Assume an algorithm A exists that meets the three criteria: atomicity, availability, and partition tolerance. We construct an execution of A in which there exists a request that returns an inconsistent response. The methodology is similar to proofs in Attiya et al. [1] and Lynch [8]. Assume that the network consists of at least two nodes. Thus it can be divided into two disjoint, non-empty sets: $\{G_1, G_2\}$. The basic idea of the proof is to assume that all messages between G_1 and G_2 are lost. Then if a *write* occurs in G_1, and later a *read* occurs in G_2, then the *read* operation cannot return the results of the earlier *write* operation.

[5] Brewer pointed out in the talk that partitions of one node are irrelevant: they are equivalent to that node failing. However restricting our attention to partitions containing only components of size greater than one does not change any of the results in this note.

More formally, let v_0 be the initial value of the atomic object. Let α_1 be the prefix of an execution of A in which a single *write* of a value not equal to v_0 occurs in G_1, ending with the termination of the *write* operation. Assume that no other client requests occur in either G_1 or G_2. Further, assume that no messages from G_1 are received in G_2, and no messages from G_2 are received in G_1. We know that this *write* completes, by the availability requirement. Similarly, let α_2 be the prefix of an execution in which a single *read* occurs in G_2, and no other client requests occur, ending with the termination of the *read* operation. During α_2 no messages from G_2 are received in G_1, and no messages from G_1 are received in G_2. Again we know that the *read* returns a value by the availability requirement. The value returned by this execution must be v_0, as no *write* operation has occurred in α_2.

Let α be an execution beginning with α_1 and continuing with α_2. To the nodes in G_2, α is indistinguishable from α_2, as all the messages from G_1 to G_2 are lost (in both α_1 and α_2, which together make up α), and α_1 does not include any client requests to nodes in G_2. Therefore in the α execution, the *read* request (from α_2) must still return v_0. However the *read* request does not begin until after the *write* request (from α_1) has completed. This therefore contradicts the atomicity property, proving that no such algorithm exists. ■

Corollary 1.1 *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability, in all fair executions,*
- *Atomic consistency, in fair executions in which* no *messages are lost.*

Proof: The main idea is that in the asynchronous model an algorithm has no way of determining whether a message has been lost, or has been arbitrarily delayed in the transmission channel. Therefore if there existed an algorithm that guaranteed atomic consistency in executions in which no messages were lost, then there would exist an algorithm that guaranteed atomic consistency in all executions. This would violate Theorem 1.

More formally, assume for the sake of contradiction that there exists an algorithm A that always terminates, and guarantees atomic consistency in fair executions in which all messages are delivered. Further, Theorem 1 implies that A does not guarantee atomic consistency in all fair executions, so there exists some fair execution α of A in which some response is not atomic.

At some finite point in execution α, the algorithm A returns a response that is not atomic. Let α' be the prefix of α ending with the invalid response. Next, extend α' to a fair execution α'', in which all messages are delivered. The execution α'' is now a fair execution in which all messages are delivered. However this execution is not atomic. Therefore no such algorithm A exists. ■

3.2 Solutions in the Asynchronous Model

While it is impossible to provide all three properties: atomicity, availability, and partition tolerance, any two of these three properties can be achieved.

3.2.1 Atomic, Partition Tolerant

If availability is not required, then it is easy to achieve atomic data and partition tolerance. The trivial system that ignores all requests meets these requirements. However we can provide a stronger liveness criterion: if all the messages in an execution are delivered, the system is available and all operations terminate. A simple centralized algorithm meets these requirements: a single designated node maintains the value of an object. A node receiving a request forwards the request to the designated node, which sends a response. When an acknowledgment is received, the node sends a response to the client.

Many distributed databases provide this type of guarantee, especially algorithms based on distributed locking or quorums: if certain failure patterns occur, then the liveness condition is weakened and the service no longer returns responses. If there are no failures, then liveness is guaranteed.

3.2.2 Atomic, Available

If there are no partitions, it is clearly possible to provide atomic, available data. In fact, the centralized algorithm described in Section 3.2.1 meets these requirements. Systems that run on intranets and LANs are an example of these types of algorithms.

3.2.3 Available, Partition Tolerant

It is possible to provide high availability and partition tolerance, if atomic consistency is not required. If there are no consistency requirements, the service can trivially return v_0, the initial value, in response to every request.

However it is possible to provide weakened consistency in an available, partition tolerant setting. Web caches are one example of a weakly consistent network. In Section 4.3 we consider some of the weaker consistency conditions that are possible.

4 Partially Synchronous Networks

4.1 Partially Synchronous Model

The most obvious way to try to circumvent the impossibility result of Theorem 1 is to realize that in the real world, most networks are not purely asynchronous. If you allow each node in the network to have a clock, it is possible to build a more powerful service.

For the rest of this paper, we will discuss a partially synchronous model in which every node has a clock, and all clocks increase at the same rate. However, the clocks themselves are not synchronized, in that they may display different values at the same real time. In effect, the clocks act as timers: local state variables that the processes can observe to measure how much time has passed. A local timer can be used to schedule an action to occur a certain interval of time after some other event. Furthermore, assume that every message is either delivered within a given, known time: t_{msg}, or it is lost. Also, every node processes a received message within a given, known time: t_{local}, and local processing takes zero time. This can be formalized as a special case of the General Timed Automata model defined by Lynch [9].

4.2 Impossibility Result

It is still impossible to have an always available, atomic data object when arbitrary messages may be lost, even in the partially synchronous model. That is, the following analogue of Theorem 1 holds:

Theorem 2 *It is impossible in the partially synchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability*
- *Atomic consistency*

in all executions (even those in which messages are lost).

Proof: This proof is rather similar to the proof of Theorem 1. We will follow the same methodology: divide the network into two components, $\{G_1, G_2\}$,

and construct an admissable execution in which a *write* happens in one component, followed by a *read* operation in the other component. This *read* operation can be shown to return inconsistent data.

More formally, construct execution α_1 as before in Theorem 1: a single *write* request and acknowledgment occur in G_1, and all messages between the two components, $\{G_1, G_2\}$, are lost. We will construct the second execution, α_2', slightly differently. Let α_2' be an execution that begins with a long interval of time during which no client requests occur. This interval must be at least as long as the entire duration of α_1. Then append to α_2' the events of α_2, as defined above in Theorem 1: a single *read* request and response in G_2, again assuming all messages between the two components are lost. Finally, construct α by superimposing the two executions α_1 and α_2'. The long interval of time in α_2 ensures that the *write* request completes before the *read* request begins. However, as in Theorem 1, the *read* request returns the initial value, rather than the new value written by the *write* request, violating atomic consistency. ■

4.3 Solutions in the Partially Synchronous Model

In the partially synchronous model, however, the analogue of Corollary 1.1 does not hold. The proof of this corollary does in fact depend on nodes being unaware of when a message is lost. There are partially synchronous algorithms that will return atomic data when all messages in an execution are delivered (i.e., there are no partitions), and will only return inconsistent (and, in particular, stale) data when messages are lost. One example of such an algorithm is the centralized protocol described in Section 3.2.1, modified to time-out lost messages. On a *read* (or *write*) request, a message is sent to the central node. If a response from the central node is received, then the node delivers the requested data (or an acknowledgment). If no response is received within $2 * t_{msg} + t_{local}$, then the node concludes that the message was lost. The client is then sent a response: either the best known value of the local node, or an acknowledgment. In this case, atomic consistency may be violated.

4.4 Weaker Consistency Conditions

While it is useful to guarantee that atomic data will be returned in executions in which all messages are delivered (within some time bound), it is equally important to specify what happens in executions in which some of the messages are lost. In this section, we will discuss one possible weaker con-

sistency condition that would allow stale data to be returned when there are partitions, yet still place formal requirements on the quality of the stale data returned. This consistency guarantee will require availability and atomic consistency in executions in which no messages are lost, and is therefore impossible to guarantee in the asynchronous model as a result of Corollary 1.1.

In the partially synchronous model it often makes sense to base guarantees on how long an algorithm has had to rectify a situation. This consistency model ensures that if messages are delivered, then eventually some notion of atomicity is restored.

In an atomic execution, we would define a partial order of the *read* and *write* operations, and then require that if one operation begins after another one ends, the former does not precede the latter in the partial order. We will define a weaker guarantee, Delayed-t consistency, that defines a partial order in a similar manner, but only requires that one operation not precede another if there was an interval between the operations in which all messages were delivered.

Definition 3 *A timed execution, α, of a read-write object is* Delayed-t Consistent *if:*

1. *P is a partial order that orders all* write *operations, and orders all* read *operations with respect to the* write *operations.*

2. *The value returned by every* read *operation is exactly the one written by the previous* write *operation in P (or the initial value, if there is no such previous* write *in P).*

3. *The order in P is consistent with the order of* read *and* write *requests submitted at each node.*

4. *(Atomicity) If all messages in the execution are delivered, and an operation θ completes before an operation ϕ begins, then ϕ does not precede θ in the partial order P,*

5. *(Weakly Consistent) Assume there exists an interval of time longer than t in which no messages are lost. Further, assume an operation, θ, completes before the interval begins, and another operation, ϕ, begins after the interval ends. Then ϕ does not precede θ in the partial order P.*

This guarantee allows for some stale data when messages are lost, but provides a time limit on how long this inconsistency can continue, once the partition heals. This is related to the idea of eventually consistent data, as described in Fekete et al. [3]; here, however, we want an explicit time bound on how long it will take for the data to become consistent.

A variant of the centralized algorithm described in Section 4.3 is Delayed-t consistent. Assume node C is the centralized node. The algorithm behaves as follows:

- *read* at node A:
  1. A sends a request to C for the most recent value.
  2. If A receives a response from C, save the value and send it to the client.
  3. If A concludes that a message was lost (i.e. a timeout occurs), then return the value with the highest sequence number received from C (see below), or the initial-value (if no value has yet been received from C).
- *write* at A:
  1. A sends a message to C with the new value.
  2. If A receives an acknowledgement from C, then A sends an acknowledgement to the client, and stops.
  3. If A concludes a message was lost (i.e. a timeout occurs), then A sends an acknowledgement to the client.
  4. If A has not yet received an acknowledgement from C, then A sends a message to C with the new value.
  5. If A concludes a message was lost (i.e. a timeout occurs), A repeats step 4 within $t - 4 * t_{\text{timeout}}$ seconds.
- New value is received at C:
  1. C increments its sequence number by 1.
  2. C sends out the new value and the sequence number to every node.
  3. If C concludes a message was lost (i.e. a timeout occurs), then C resends the value and sequence number to the missing node within time $t - 2 * t_{\text{timeout}}$ seconds.

4. Repeat step 3 until every node has acknowledged the value.

Theorem 4 *The modified centralized algorithm is Delayed-t consistent.*

Proof: The ordering of *write* operations is the order in which the centralized node is notified of the client *write* operations. The centralized node assigns each a sequence number, and this determines the total ordering of *write* operations. Each *read* operation is sequenced after the *write* operation whose value it returns. If all the messages in an execution are delivered, this algorithm is atomic: the centralized node serializes requests, and ensures that the partial order is correctly respected. If an operation has completed, then the centralized node has already been notified, and therefore no operation that begins later will precede the completed operation in the partial order.

If, on the other hand, some messages are lost, the resulting executions might not be atomic. Assume a *write* occurs at A_w, after which an interval of time passes in which all messages are delivered. By the end of the interval, A_w will have notified the central node of the new value, and the central node will have notified all other nodes of this new value. Therefore no later operation after this interval will precede the *write* operation.

Assume, then, that a *read* operation occurs at B_r, after which an interval of time passes in which all messages are delivered. B_r must have received its value from the central node, C, or from an earlier *write* operation at B_r. In the former case, by the end of this interval C will have ensured that every node has received a value at least as recent as the one returned by B_r. In the latter case, by the end of the interval B_r will have sent the value to C and C will have forwarded it to every other node. Therefore no later operation after this interval will precede the *read* operation. ■

5 Conclusion

In this note, we have shown that it is impossible to reliably provide atomic, consistent data when there are partitions in the network. It is feasible, however, to achieve any two of the three properties: consistency, availability, and partition tolerance. In an asynchronous model, when no clocks are available, the impossibility result is fairly strong: it is impossible to provide consistent data, even allowing stale data to be returned when messages are lost. However in partially synchronous models it is possible to achieve a practical compromise between consistency and availability. In particular, most real-world systems today are forced to settle with returning "most of

the data, most of the time." Formalizing this idea and studying algorithms for achieving it is an interesting subject for future theoretical research.

References

[1] Attiya, Bar-Noy, Dolev, Koller, Peleg, and Reischuk. Achievable cases in an asynchronous environment. In *28th Annual Symposium on Foundations of Computer Science*, pages 337–346, Los Angeles, California, October 1987.

[2] Eric A. Brewer. Towards robust distributed systems. (Invited Talk) *Principles of Distributed Computing*, Portland, Oregon, July 2000.

[3] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1):113–156, June 1999.

[4] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[5] Leslie Lamport. On interprocess communication – parts i and ii. *Distributed Computing*, 1(2):77–101, April 1986.

[6] Nancy Lynch. *Distributed Algorithms*, pages 397–350. Morgan Kaufman, 1996.

[7] Nancy Lynch. *Distributed Algorithms*, pages 199–231. Morgan Kaufman, 1996.

[8] Nancy Lynch. *Distributed Algorithms*, page 581. Morgan Kaufman, 1996.

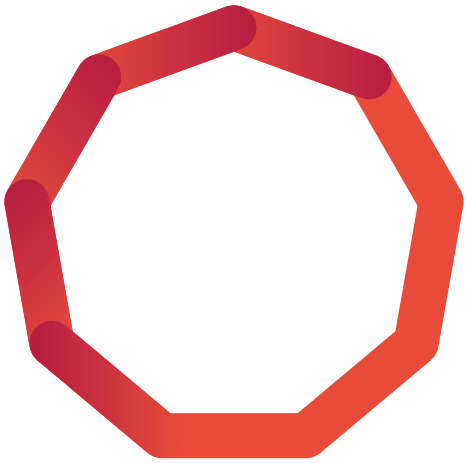
[9] Nancy Lynch. *Distributed Algorithms*, pages 735–770. Morgan Kaufman, 1996.

Chapter 7

MapReduce: Simplified Data Processing on Large Clusters

Authors:
Jeffrey Dean, Sanjay Ghemawat

Publication year: 2004



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle$`URL`, `1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL`, `total count`$\rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle$`target`, `source`$\rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle$`target`, $list$(`source`)$\rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname`, `term vector`$\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname`, `term vector`$\rangle$ pair.

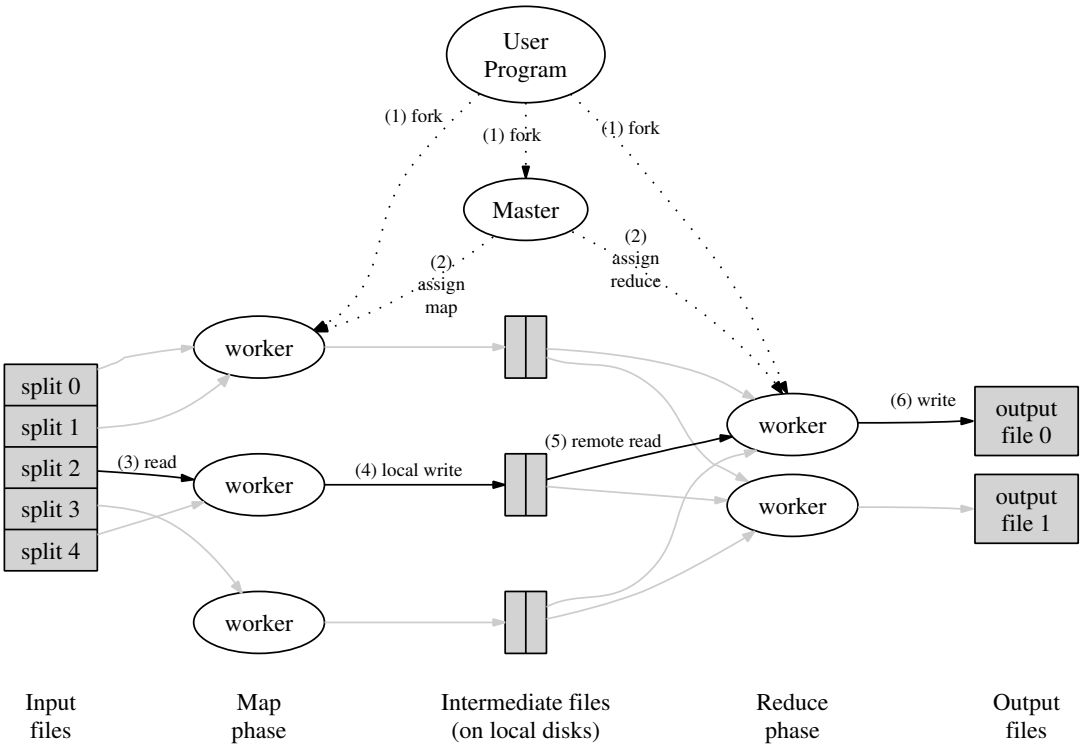


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle \texttt{word}, \texttt{document ID} \rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle \texttt{word}, list(\texttt{document ID}) \rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle \texttt{key}, \texttt{record} \rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \mathbf{mod} R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M and $e(R_2)$ may have read the output produced by a different execution of M.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key) \bmod R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey)) \bmod R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

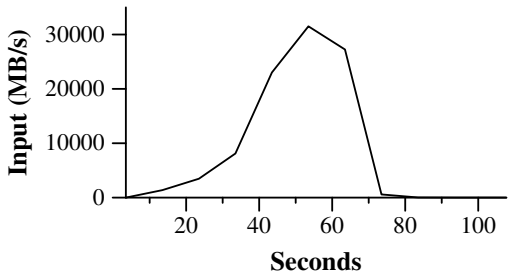


Figure 2: Data transfer rate over time

5.2 Grep

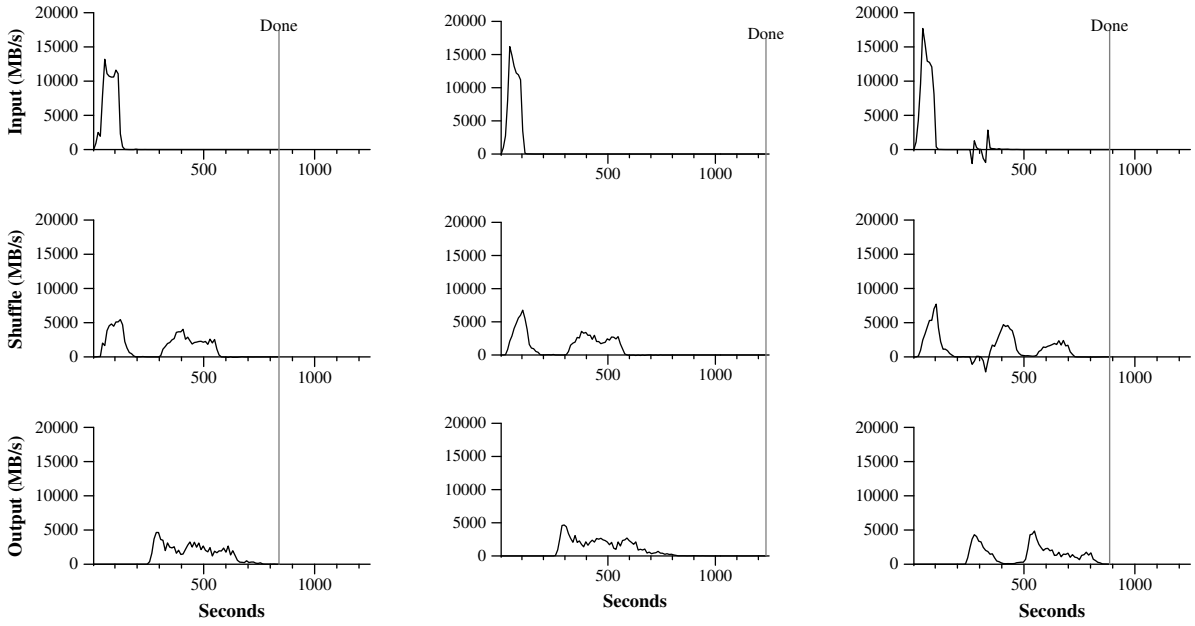
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

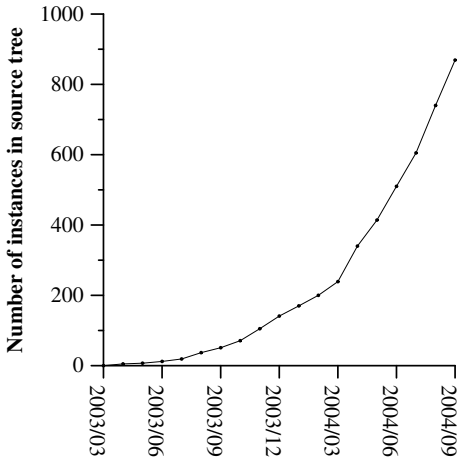


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;

      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

Chapter 8

Paxos Made Live - An Engineering Perspective

Authors:
Tushar Chandra, Robert Griesemer, Joshua Redstone

Publication year: 2007

Paxos Made Live - An Engineering Perspective

Tushar Chandra
Robert Griesemer
Joshua Redstone

June 20, 2007

Abstract

We describe our experience in building a fault-tolerant data-base using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system.

1 Introduction

It is well known that fault-tolerance on commodity hardware can be achieved through replication [17, 18]. A common approach is to use a consensus algorithm [7] to ensure that all replicas are mutually consistent [8, 14, 17]. By repeatedly applying such an algorithm on a sequence of input values, it is possible to build an identical log of values on each replica. If the values are operations on some data structure, application of the same log on all replicas may be used to arrive at mutually consistent data structures on all replicas. For instance, if the log contains a sequence of database operations, and if the same sequence of operations is applied to the (local) database on each replica, eventually all replicas will end up with the same database content (provided that they all started with the same initial database state).

This general approach can be used to implement a wide variety of fault-tolerant primitives, of which a fault-tolerant database is just an example. As a result, the consensus problem has been studied extensively over the past two decades. There are several well-known consensus algorithms that operate within a multitude of settings and which tolerate a variety of failures. The Paxos consensus algorithm [8] has been discussed in the theoretical [16] and applied community [10, 11, 12] for over a decade.

We used the Paxos algorithm ("Paxos") as the base for a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the literature and some not.

- The fault-tolerant algorithms community is accustomed to proving short algorithms (one page of pseudo code) correct. This approach does not scale to a system with thousands of lines of code. To gain confidence in the "correctness" of a real system, different methods had to be used.

- Fault-tolerant algorithms tolerate a limited set of carefully selected faults. However, the real world exposes software to a wide variety of failure modes, including errors in the algorithm, bugs in its

©ACM 2007. This is a minor revision of the work that will be published in the proceedings of ACM PODC 2007.

implementation, and operator error. We had to engineer the software and design operational procedures to robustly handle this wider set of failure modes.

- A real system is rarely specified precisely. Even worse, the specification may change during the implementation phase. Consequently, an implementation should be malleable. Finally, a system might "fail" due to a misunderstanding that occurred during its specification phase.

This paper discusses a selection of the algorithmic and engineering challenges we encountered in moving Paxos from theory to practice. This exercise took more R&D efforts than a straightforward translation of pseudo-code to C++ might suggest.

The rest of this paper is organized as follows. The next two sections expand on the motivation for this project and describe the general environment into which our system was built. We then provide a quick refresher on Paxos. We divide our experiences into three categories and discuss each in turn: algorithmic gaps in the literature, software engineering challenges, and unexpected failures. We conclude with measurements of our system, and some broader observations on the state of the art in our field.

2 Background

Chubby [1] is a fault-tolerant system at Google that provides a distributed locking mechanism and stores small files. Typically there is one Chubby instance, or "cell", per data center. Several Google systems – such as the Google Filesystem (GFS) [4] and Bigtable [2] – use Chubby for distributed coordination and to store a small amount of metadata.

Chubby achieves fault-tolerance through replication. A typical Chubby cell consists of five replicas, running the same code, each running on a dedicated machine. Every Chubby object (e.g., a Chubby lock, or file) is stored as an entry in a database. It is this database that is replicated. At any one time, one of these replicas is considered to be the "master".

Chubby clients (such as GFS and Bigtable) contact a Chubby cell for service. The master replica serves all Chubby requests. If a Chubby client contacts a replica that is not the master, the replica replies with the master's network address. The Chubby client may then contact the master. If the master fails, a new master is automatically elected, which will then continue to serve traffic based on the contents of its local copy of the replicated database. Thus, the replicated database ensures continuity of Chubby state across master failover.

The first version of Chubby was based on a commercial, third-party, fault-tolerant database; we will refer to this database as "3DB" for the rest of this paper. This database had a history of bugs related to replication. In fact, as far as we know, the replication mechanism was not based on a proven replication algorithm and we do not know if it is correct. Given the history of problems associated with that product and the importance of Chubby, we eventually decided to replace 3DB with our own solution based on the Paxos algorithm.

3 Architecture outline

Figure 1 illustrates the architecture of a single Chubby replica. A fault-tolerant replicated log based on the Paxos algorithm sits at the bottom of the protocol stack. Each replica maintains a local copy of the log. The Paxos algorithm is run repeatedly as required to ensure that all replicas have identical sequences of entries in their local logs. Replicas communicate with each other through a Paxos-specific protocol.

The next layer is a fault-tolerant replicated database which includes a local copy of the database at each replica. The database consists of a local *snapshot* and a *replay*-log of database operations. New database operations are submitted to the replicated log. When a database operation appears at a replica, it is applied on that replica's local database copy.

Finally, Chubby uses the fault-tolerant database to store its state. Chubby clients communicate with a single Chubby replica through a Chubby-specific protocol.

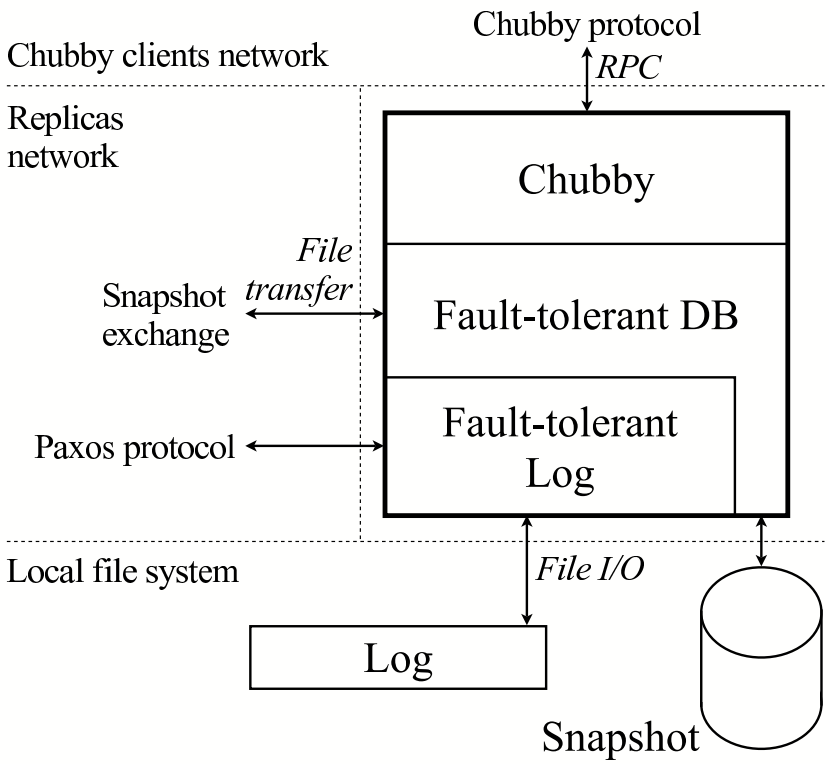


Figure 1: A single Chubby replica.

We devoted effort to designing clean interfaces separating the Paxos framework, the database, and Chubby. We did this partly for clarity while developing this system, but also with the intention of reusing the replicated log layer in other applications. We anticipate future systems at Google that seek fault-tolerance through replication. We believe that a fault-tolerant log is a powerful primitive on which to build such systems.

Our fault-tolerant log's API is depicted in Figure 2. It contains a call to *submit* a new value to the log. Once a submitted value enters the fault-tolerant log, our system invokes a callback to the client application at each replica and passes the submitted value.

Our system is multi-threaded and multiple values can be submitted concurrently on different threads. The replicated log does not create its own threads but can be invoked concurrently by any number of threads. This approach to threading helps in testing the system, as we will show in more detail later in the paper.

4 On Paxos

In this section we give an informal overview of the basic Paxos algorithm and outline how to chain together multiple executions of it (Multi-Paxos). We refer the reader to the literature for more formal descriptions and correctness proofs [8, 9, 16]. Readers who are familiar with Paxos may skip directly to the next section.

4.1 Paxos Basics

Paxos is a consensus algorithm executed by a set of processes, termed *replicas*, to agree on a single value in the presence of failures. Replicas may crash and subseqently recover. The network may drop messages

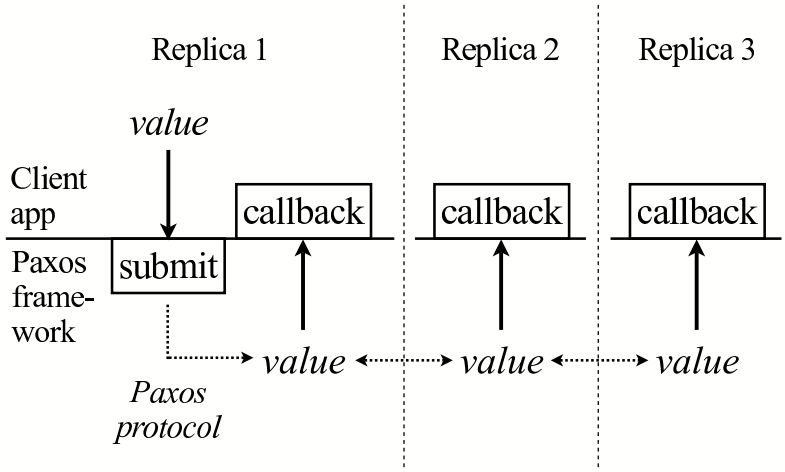


Figure 2: API for fault-tolerant log.

between replicas. Replicas have access to persistent storage that survives crashes. Some replicas may *submit* values for consensus. If eventually a majority of the replicas run for long enough without crashing and there are no failures, all running replicas are guaranteed to *agree* on one of the values that was submitted. In our system, the value to be agreed upon is the next entry in a (replicated) log as described in the introduction.

The algorithm consists of three phases, which may be repeated (because of failures):

1. Elect a replica to be the *coordinator*.
2. The coordinator selects a value and broadcasts it to all replicas in a message called the *accept* message. Other replicas either *acknowledge* this message or *reject* it.
3. Once a majority of the replicas *acknowledge* the coordinator, consensus has been reached, and the coordinator broadcasts a *commit* message to notify replicas.

To provide some intuition about how the algorithm works, consider first the case in which there is only a single coordinator and no failures. Consensus is reached once a majority of replicas receive the accept message from the coordinator and acknowledge it. Subsequently, if any minority of the replicas fail, we are still guaranteed that at least one replica will be alive that received the consensus value.

In reality the coordinator may fail. Paxos does not require that only one replica act as coordinator at a time. Multiple replicas may decide to become coordinators and execute the algorithm at any time. Typically the system is engineered to limit coordinator turnover, as it can delay reaching consensus.

This flexible election policy means there may be multiple replicas who simultaneously believe they are the coordinator. Further, these coordinators may select different values. Paxos ensures consensus can be reached on a single value (it can be from any coordinator) by introducing two extra mechanisms: 1) assigning an ordering to the successive coordinators; and 2) restricting each coordinator's choice in selecting a value.

Ordering the coordinators allows each replica to distinguish between the current coordinator and previous coordinators. In this way, replicas can reject messages from old coordinators and prevent them from disrupting consensus once it is reached. Paxos orders the coordinators by assigning them an increasing *sequence number* as follows. Each replica keeps track of the most recent sequence number it has seen so far. When a replica wants to become coordinator, it generates a unique[1] sequence number higher than any it has seen,

[1]For example, in a system with n replicas, assign each replica r a unique id i_r between 0 and $n-1$. Replica r picks the smallest sequence number s larger than any it has seen such that $s \ mod \ n = i_r$.

and broadcasts it to all replicas in a *propose* message. If a majority of replicas reply and indicate they have not seen a higher sequence number, then the replica acts as a coordinator. These replies are called *promise* messages since replicas promise henceforth to reject messages from old coordinators. This propose/promise message exchange constitutes step 1 listed above.

Once consensus is reached on a value, Paxos must force future coordinators to select that same value in order to ensure continued agreement. To guarantee this, the promise messages from replicas include the most recent value they have heard, if any, along with the sequence number of the coordinator from whom they heard it. The new coordinator chooses the value from the most recent coordinator. If none of the promise messages contain a value, the coordinator is free to choose a submitted value.

The reasoning why this works is subtle, but proceeds roughly as follows. The new coordinator requires a response to the propose message from a majority of replicas. Therefore, if consensus was achieved by a previous coordinator, the new coordinator is guaranteed to hear about the value decided upon from at least one replica. By induction, that value will have the highest sequence number of all responses received, and so will be selected by the new coordinator.

4.2 Multi-Paxos

Practical systems use Paxos as a building block to achieve consensus on a *sequence of values*, such as in a replicated log. The simple way to implement this is to repeatedly execute the Paxos algorithm. We term each execution an *instance* of Paxos. We refer to *submitting* a value to Paxos (or equivalently, to the log) to mean executing an instance of Paxos while submitting that value.

In Multi-Paxos some slow (*lagging*) replicas might not have participated in recent Paxos instances. We use a *catch-up* mechanism to enable lagging replicas to catch up with *leading* replicas.

Each replica maintains a locally persistent log to record all Paxos actions. When a replica crashes and subsequently recovers, it replays the persistent log to reconstruct its state prior to crashing. Replicas also use this log when helping lagging replicas to catch up. The Paxos algorithm as described thus far requires all message senders to log their state before sending messages – thus the algorithm requires a sequence of five writes (for each of the propose, promise, accept, acknowledgment, and commit messages) to disk on its critical path. Note that all writes have to be flushed to disk immediately before the system can proceed any further. In a system where replicas are in close network proximity, disk flush time can dominate the overall latency of the implementation.

There is a well-known optimization to reduce the number of messages involved by chaining together multiple Paxos instances [9]. Propose messages may be omitted if the coordinator identity does not change between instances. This does not interfere with the properties of Paxos because any replica at any time can still try to become coordinator by broadcasting a propose message with a higher sequence number. In order to avail itself of this optimization, a Multi-Paxos algorithm may be designed to pick a coordinator for long periods of time, trying not to let the coordinator change. We refer to this coordinator as the *master*. With this optimization, the Paxos algorithm only requires a single write to disk per Paxos instance on each replica, executed in parallel with each other. The master writes to disk immediately after sending its accept message and other replicas write to disk prior to sending their acknowledge message.

In order to get additional throughput in a concurrent system, it is possible to batch a collection of values submitted by different application threads into a single Paxos instance.

5 Algorithmic challenges

While the core Paxos algorithm is well-described, implementing a fault-tolerant log based on it is a non-trivial endeavor. Some of the complications are due to imperfections found in the real world (such as hard disk failures, or finite resources), and some are due to additional requirements (for instance, "master leases"). Many of these challenges have algorithmic solutions that are intimately connected with the core Paxos algorithm. In the following we describe a number of mechanisms that we introduced.

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file[2]. The latter may be indistinguishable from a new replica with an empty disk – we detect this case by having a new replica leave a marker in GFS after start-up. If this replica ever starts again with an empty disk, it will discover the GFS marker and indicate that it has a corrupted disk.

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

This mechanism enables the following optimization to improve the latency of the system. Since the system can now deal with occasional disk corruption, under some circumstances it may be acceptable not to flush writes to disk immediately[3]. While we have considered schemes to exploit this observation, we have not implemented them yet.

5.2 Master leases

When the basic Paxos algorithm is used to implement a replicated data structure, reads of the data structure require executing an instance of Paxos. This serializes the read with respect to updates and ensures that the *current* state is read. In particular, read operations cannot be served out of the master's copy of the data structure because it is possible that other replicas have elected another master and modified the data structure without notifying the old master. In this case, the read operation at the master runs the risk of returning stale data. Since read operations usually comprise a large fraction of all operations, serializing reads through Paxos is expensive.

The workaround is to implement *master leases* [5] with the following semantics: as long as the master has the lease, it is guaranteed that other replicas cannot successfully submit values to Paxos. Thus a master with the lease has up-to-date information in its local data structure which can be used to serve a read operation purely locally. By making the master attempt to renew its lease before it expires we can ensure that a master has a lease most of the time. With our system, masters successfully maintain leases for several days at a time.

In our implementation, all replicas implicitly grant a lease to the master of the previous Paxos instance and refuse to process Paxos messages from any other replica while the lease is held. The master maintains a shorter timeout for the lease than the replicas – this protects the system against clock drift. The master periodically submits a dummy "heartbeat" value to Paxos to refresh its lease.

The Multi-Paxos optimization exhibits the following stability problem when there are intermittent network outages. When a master temporarily disconnects, Paxos will elect a new master. The new master will maintain a fixed sequence number across instances of Paxos. In the mean time, when the disconnected old master tries to run the Paxos algorithm, if it manages to connect with another replica, it may increase its sequence number. When it reconnects, it may have a higher sequence number than the new master and be able to replace the new master. Later it may disconnect again, and the cycle can repeat itself.

This behavior is undesirable as Chubby master changes have a negative impact on some of its users. Furthermore, this behavior can degenerate into rapid master changes in a network with poor connectivity.

[2]This mechanism will not detect files that roll back to an old state. We believe this is an unlikely scenario and chose not to handle it explicitly. Our distributed checksum mechanism, described later, may detect this type of problem.

[3]For example, if the operating system and the hardware underlying each replica rarely fail and failures at different replicas are independent of each other, it is possible to modify our system so it does not need to flush writes to disk.

In our implementation the master periodically boosts its sequence number by running a full round of the Paxos algorithm, including sending propose messages[4]. Boosting with the right frequency avoids this type of master churn in most cases.

Note that it is possible to extend the concept of leases to all replicas. This will allow any replica with a lease to serve read requests from its local data structure. This extended lease mechanism is useful when read traffic significantly exceeds write traffic. We have examined algorithms for replica leases, but have not implemented them yet.

5.3 Epoch numbers

Requests (by a Chubby client) submitted to a Chubby cell are directed to the current Chubby master replica. From the time when the master replica receives the request to the moment the request causes an update of the underlying database, the replica may have lost its master status. It may even have lost master status and regained it again. Chubby requires an incoming request to be aborted if mastership is lost and/or re-acquired during the handling of the request. We needed a mechanism to reliably detect master turnover and abort operations if necessary.

We solved this problem by introducing a global epoch number with the following semantics. Two requests for the epoch number at the master replica receive the same value iff that replica was master continuously for the time interval between the two requests. The epoch number is stored as an entry in the database, and all database operations are made conditional on the value of the epoch number.

5.4 Group membership

Practical systems must be able to handle changes in the set of replicas. This is referred to as the group membership problem in the literature [3]. Some Paxos papers point out that the Paxos algorithm itself can be used to implement group membership [8]. While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos, disk corruptions, etc. Unfortunately the literature does not spell this out, nor does it contain a proof of correctness for algorithms related to group membership changes using Paxos. We had to fill in these gaps to make group membership work in our system. The details – though relatively minor – are subtle and beyond the scope of this paper.

5.5 Snapshots

As described thus far, the repeated application of a consensus algorithm to create a replicated log will lead to an ever growing log. This has two problems: it requires unbounded amounts of disk space; and perhaps worse, it may result in unbounded recovery time since a recovering replica has to replay a potentially long log before it has fully caught up with other replicas. Since the log is typically a sequence of operations to be applied to some data structure, and thus implicitly (through replay) represents a persistent form of that data structure, the problem is to find an alternative persistent representation for the data structure at hand. An obvious mechanism is to persist – or *snapshot* – the data structure directly, at which point the log of operations leading to the current state of the data structure is no longer needed. For example, if the data structure is held in memory, we take a snapshot by serializing it on disk. If the data structure is kept on disk, a snapshot may just be an on-disk copy of it.

By itself, the Paxos framework does not know anything about the data structure we are trying to replicate; its only concern is the consistency of the replicated log. It is the particular application using the Paxos framework that has all the knowledge about the replicated data structure. Thus the application must be responsible for taking snapshots. Our framework provides a mechanism that allows client applications, e.g. our fault-tolerant database, to inform the framework that a snapshot was taken; the client application is free to take a snapshot at any point. When the Paxos framework is informed about a snapshot, it will truncate its log by deleting log entries that precede the snapshot. Should the replica fail, during subsequent recovery

[4]In a loaded system, under one percent of the Paxos instances run the full Paxos algorithm.

it will simply install the latest snapshot and then replay the truncated log to rebuild its state. Snapshots are not synchronized across replicas; each replica independently decides when to create a snapshot.

This mechanism appears straightforward at first and is mentioned briefly in the literature [8]. However, it introduces a fair amount of complexity into the system: the persistent state of a replica now comprises a log *and* a snapshot that have to be maintained consistently. The log is fully under the framework's control, while the snapshot format is application-specific. Some aspects of the snapshot machinery are of particular interest:

- The snapshot and log need to be mutually consistent. Each snapshot needs to have information about its contents relative to the fault-tolerant log. In our framework we introduced the concept of a *snapshot handle* for this purpose. The snapshot handle contains all the Paxos-specific information related to a particular snapshot. When creating a snapshot (which is under control of the application) the corresponding snapshot handle (provided by the framework) needs to be stored by the application as well. When recovering a snapshot, the application must return the snapshot handle to the framework, which in turn will use the information in the handle to coordinate the snapshot with the log.

  Note that the handle is really a snapshot of the Paxos state itself. In our system, it contains the Paxos instance number corresponding to the (log) snapshot and the group membership at that point.

- Taking a snapshot takes time and in some situations we cannot afford to freeze a replica's log while it is taking a snapshot. In our framework, taking a snapshot is split into three phases. First, when the client application decides to take a snapshot, it requests a snapshot handle. Next, the client application takes its snapshot. It may block the system while taking the snapshot, or – more likely – spawn a thread that takes a snapshot while the replica continues to participate in Paxos. The snapshot must correspond to the client state at the log position when the handle was obtained. Thus if the replica continues to participate in Paxos while taking a snapshot, special precautions may have to be taken to snapshot the client's data structure while it is actively updated.[5] Finally, when the snapshot has been taken, the client application informs the framework about the snapshot and passes the corresponding snapshot handle. The framework then truncates the log appropriately.

- Taking a snapshot may fail. Our framework only truncates the log when it is informed that a snapshot has been taken and has received the corresponding snapshot handle. Thus, as long as the client application does not inform the framework, from the framework's viewpoint, no snapshot has been taken. This allows the client application to verify a snapshot's integrity and discard it if necessary. If there is a problem with the snapshot, the client doesn't ask the framework to truncate its log. A client application may even attempt to take several snapshots at the same time using this mechanism.

- While in catch-up, a replica will attempt to obtain missing log records. If it cannot obtain them (because no replica has old-enough log entries readily available), the replica will be told to obtain a snapshot from another replica. This snapshot's handle contains information about the Paxos instance up to which the snapshot captured the state. Once the snapshot has been received and installed, under most circumstances the lagging replica will be close to the leading replica. In order to completely catch-up, the lagging replica asks for and receives the remaining log records from the leading replica to bring it fully up-to-date.

  Note that a leading replica may even create a new snapshot while a lagging replica is installing an older snapshot – in a fault-tolerant system this cannot be avoided. In this scenario, the lagging replica may not be able to obtain any outstanding log records because the snapshot provider (and any other replicas) may have moved ahead in the meantime. The lagging replica will need to obtain a more recent snapshot.

[5] Our first implementation of the fault-tolerant database blocked the system very briefly while making an in-memory copy of the (small) database. It then stored the copied data on disk via a separate thread. Subsequently we implemented virtually pause-less snapshots. We now use a "shadow" data structure to track updates while the underlying database is serialized to disk.

Furthermore, the leading replica may fail after sending its snapshot. The catch-up mechanism must be able to recover from such problems by having the lagging replica contact another leading replica.

- We needed a mechanism to locate recent snapshots. Some applications may choose to transfer snapshots directly between leading and lagging replicas while others may ask a lagging replica to look up a snapshot on GFS. We implemented a general mechanism that allows an application to pass snapshot location information between leading and lagging replicas.

5.6 Database transactions

The database requirements imposed by Chubby are simple: the database needs to store key-value pairs (with keys and values being arbitrary strings), and support common operations such as `insert`, `delete`, `lookup`, an atomic `compare and swap` (`cas`), and iteration over all entries. We implemented a log-structured design using a snapshot of the full database, and a log of database operations to be applied to that snapshot. The log of operations is the Paxos log. The implementation periodically takes a snapshot of the database state and truncates the log accordingly.

The `cas` operation needed to be atomic with respect to other database operations (potentially issued by a different replica). This was easily achieved by submitting all `cas`-related data as a single "value" to Paxos. We realized that we could extend this mechanism to provide transaction-style support without having to implement true database transactions. We describe our solution in more detail because we believe it to be useful in other contexts.

Our implementation hinges around a powerful primitive which we call `MultiOp`. All other database operations except for iteration are implemented as a single call to `MultiOp`. A `MultiOp` is applied atomically and consists of three components:

1. A list of tests called **guard**. Each test in `guard` checks a single entry in the database. It may check for the absence or presence of a value, or compare with a given value. Two different tests in the guard may apply to the same or different entries in the database. All tests in the guard are applied and `MultiOp` returns the results. If all tests are true, `MultiOp` executes `t_op` (see item 2 below), otherwise it executes `f_op` (see item 3 below).

2. A list of database operations called **`t_op`**. Each operation in the list is either an `insert`, `delete`, or `lookup` operation, and applies to a single database entry. Two different operations in the list may apply to the same or different entries in the database. These operations are executed[6] if **guard** evaluates to true.

3. A list of database operations called **`f_op`**. Like `t_op`, but executed if **guard** evaluates to false.

Late in our development (and after we had implemented the database and `MultiOp`), we realized that we also needed epoch numbers to implement database operations for Chubby. With this additional requirement, all Chubby operations became associated with an epoch number and were required to fail if the Paxos epoch number changed. `MultiOp` proved useful in accomodating this new requirement. After we incorporated the Paxos epoch as a database entry, we were able to modify all previous calls to our database to include an additional guard to check for the epoch number.

6 Software Engineering

Fault-tolerant systems are expected to run continuously for long periods of time. Users are much less likely to tolerate bugs than in other systems. For instance, a layout bug exhibited by a document editor may be annoying to a user, but often it is possible to "work around" the issue, even though the bug is really at the

[6] Each `MultiOp` operation is serialized atomically with respect to other operations. The individual operations in the list are executed sequentially on the database.

core of what the software is supposed to do. A bug of similar gravity in a fault-tolerant system may make the system unusable.

We adopted several software engineering methods to give us confidence in the robustness of our implementation. We describe some of the methods we used in this section.

6.1 Expressing the algorithm effectively

Fault-tolerant algorithms are notoriously hard to express correctly, even as pseudo-code. This problem is worse when the code for such an algorithm is intermingled with all the other code that goes into building a complete system. It becomes harder to see the core algorithm, to reason about it, or to debug it when a bug is present. It also makes it difficult to change the core algorithm in response to a requirement change.

We addressed this problem by coding the core algorithm as two explicit state machines. For that purpose, we designed a simple state machine specification language and built a compiler to translate such specifications into C++. The language was designed to be terse so that a full algorithm can be rendered on a single screen. As an additional benefit, the state machine compiler also automatically generates code to log state transitions and measure code coverage to assist in debugging and testing.

We believe that choosing a specification language makes it easier to reason about and modify our state machines than an explicitly coded implementation that is intermingled with the rest of the system. This is illustrated by the following experience. Towards the end of our development of the fault-tolerant log, we had to make a fundamental change in our group membership algorithm. Prior to this change, a replica roughly went through three states. Initially it waited to join the group, then it joined the group, and finally it left the group. Once a replica left the group, it was not allowed to rejoin the group. We felt this approach was best because an intermittently failing replica would not be able to join the group and disrupt it for long. Intermittent failure turned out to be more common than originally anticipated because normal replicas exhibit intermittent failures from time to time. Thus, we needed to change the algorithm to have two states. Either a replica was in the group or it was out. A replica could switch between these two states often during the lifetime of the system. It took us about one hour to make this change and three days to modify our tests accordingly. Had we intermingled our state machines with the rest of the system, this change would have been more difficult to make.

6.2 Runtime consistency checking

The chance for inconsistencies increases with the size of the code base, the duration of a project, and the number of people working simultaneously on the same code. We used various active self-checking mechanisms such as the liberal use of `assert` statements, and explicit verification code that tests data structures for consistency.

For example, we used the following database consistency check. The master periodically submits a `checksum` request to the database log. On receipt of this request, each replica computes a checksum of its local database[7]. Since the Paxos log serializes all operations identically on all replicas, we expect all replicas to compute the same checksum. After the master completes a checksum computation, it sends its checksum to all replicas which compare the master's checksum with their computed checksum.

We have had three database inconsistency incidents thus far:

- The first incident was due to an operator error.
- We have not found an explanation for the second incident. On replaying the faulty replica's log we found that it was consistent with the other replicas. Thus it is possible that this problem was caused by a random hardware memory corruption.
- We suspect the third was due to an illegal memory access from errant code in the included codebase (which is of considerable size). To protect against this possibility in the future, we maintain a second database of checksums and double-check every database access against the database of checksums.

[7] We use a shadow datastructure to handle database operations concurrently with the checksum computation.

In all three cases manual intervention appeared to resolve the problem before it became visible to Chubby.

6.3 Testing

Given the current state of the art, it is unrealistic to prove a real system such as ours correct. To achieve robustness, the best practical solution in addition to meticulous software engineering is to test a system thoroughly. Our system was designed to be testable from the onset and now contains an extensive suite of tests. In this section we describe two tests that take the system through a long sequence of random failures and verify that it behaves as expected. Both tests can run in one of two modes:

1. **Safety mode**. In this mode, the test verifies that the system is consistent. However, the system is not required to make any progress. For example, it is acceptable for an operation to fail to complete or to report that the system is unavailable.

2. **Liveness mode**. In this mode, the test verifies that the system is consistent and is making progress. All operations are expected to complete and the system is required to be consistent.

Our tests start in safety mode and inject random failures into the system. After running for a predetermined period of time, we stop injecting failures and give the system time to fully recover. Then we switch the test to liveness mode. The purpose for the liveness test is to verify that the system does not deadlock after a sequence of failures.

One of our tests verifies the fault-tolerant log. It simulates a distributed system consisting of a random number of replicas and takes our fault-tolerant log through a random sequence of network outages, message delays, timeouts, process crashes and recoveries, file corruptions, schedule interleavings, etc. We wanted this test to be repeatable to aid in debugging. To this end, we use a random number generator to determine the schedule of failures. The seed for the random number generator is given at the beginning of the test run. We ensure that two test runs with the same random number seed are identical by running the test in a single thread to remove unwanted non-determinism from multi-threading. This is possible because the fault-tolerant log does not create its own threads and can run in a single-threaded environment (even though it normally runs in a multi-threaded environment).

Each test execution reports success or failure. If a test fails, we rerun that test with the failing random number seed and with detailed logging turned on in a debugger to determine what went wrong. This is possible because these tests are repeatable.

This test proved useful in finding various subtle protocol errors, including errors in our group membership implementation, and our modifications to deal with corrupted disks. In order to measure the strength of this test, we left some protocol bugs found during code and design reviews in the system, and verified that our test system detected these bugs. After a number of bug fixes, the test became very stable. In order to improve its bug yield, we started running this test on a farm of several hundred Google machines at a time. We found additional bugs, some of which took weeks of simulated execution time (at extremely high failure rates) to find.

Another test verifies robustness of the new Chubby system against lower-level system and hardware failures. We implemented several hooks in our fault-tolerant log to inject failures. The test randomly invokes these hooks and verifies that higher levels of the system can cope. Our hooks can be used to crash a replica, disconnect it from other replicas for a period of time or force a replica to pretend that it is no longer the master. This test found five subtle bugs in Chubby related to master failover in its first two weeks. In the same vein, we built a filesystem with hooks to programmatically inject failures and are using it to test our ability to deal with filesystem failures.

In closing we point out a challenge that we faced in testing our system for which we have no systematic solution. By their very nature, fault-tolerant systems try to mask problems. Thus they can mask bugs or configuration problems while insidiously lowering their own fault-tolerance. For example, we have observed the following scenario. We once started a system with five replicas, but misspelled the name of one of the replicas in the initial group. The system appeared to run correctly as the four correctly configured replicas

were able to make progress. Further, the fifth replica continunously ran in catch-up mode[8] and therefore appeared to run correctly as well. However in this configuration the system only tolerates one faulty replica instead of the expected two. We now have processes in place to detect this particular type of problem. We have no way of knowing if there are other bugs/misconfigurations that are masked by fault-tolerance.

6.4 Concurrency

At the onset of the project we were concerned about the problem of testing concurrent fault-tolerant code. In particular, we wanted our tests to be repeatable. As described earlier, our fault-tolerant log doesn't contain any of its own threads (even though it can handle concurrent requests on different threads). Threading is introduced at the edges of the code – where we receive calls from the networking layer. By making our tests repeatable, we were able to hunt down several obscure protocol errors during testing.

As the project progressed, we had to make several subsystems more concurrent than we had intended and sacrifice repeatability. Chubby is multi-threaded at its core, thus we cannot run repeatable tests against the complete system. Next we had to make our database multi-threaded so it could take snapshots, compute checksums and process iterators while concurrently serving database requests. Finally, we were forced to make the code that handles the local copy of the log multi-threaded as well (the exact reason why is beyond the scope of this paper).

In summary, we believe that we set ourselves the right goals for repeatability of executions by constraining concurrency. Unfortunately, as the product needs grew we were unable to adhere to these goals.

7 Unexpected failures

So far, our system has logged well over 100 machine years of execution in production. In this period we have witnessed the following unexpected failure scenarios:

- Our first release shipped with ten times the number of worker threads as the original Chubby system. We hoped this change would enable us to handle more requests. Unfortunately, under load, the worker threads ended up starving some other key threads and caused our system to time out frequently. This resulted in rapid master failover, followed by en-masse migrations of large numbers of clients to the new master which caused the new master to be overwhelmed, followed by additional master failovers, and so on.

  When this problem first appeared, the precise cause was unknown and we had to protect ourselves from a potentially dangerous bug in our system. We decided to err on the side of caution and to rollback our system to the old version of Chubby (based on 3DB) in one of our data centers. At that point, the rollback mechanism was not properly documented (because we never expected to use it), its use was non-intuitive, the operator performing the roll-back had no experience with it, and when the rollback was performed, no member of the development team was present. As a result, an old snapshot was accidentally used for the rollback. By the time we discovered the error, we had lost 15 hours of data and several key datasets had to be rebuilt.

- When we tried to upgrade this Chubby cell again a few months later, our upgrade script failed because we had omitted to delete files generated by the failed upgrade from the past. The cell ended up running with a months-old snapshot for a few minutes before we discovered the problem. This caused us to lose about 30 minutes of data. Fortunately all of Chubby's clients recovered from this outage.

- A few months after our initial release, we realized that the semantics provided by our database were different from what Chubby expected. If Chubby submitted an operation to the database, and the database lost its master status, Chubby expected the operation to fail. With our system, a replica

[8] In our implementation, a replica that is not (yet) a group member runs in catch-up mode to stay up-to-date. This allows us to keep a future group member "warm" so it can become an active member immediately after joining the group.

could be re-installed as master during the database operation and the operation could succeed. The fix required a substantial rework of the integration layer between Chubby and our framework (we needed to implement epoch numbers). `MultiOp` proved to be helpful in solving this unexpected problem – an indication that `MultiOp` is a powerful primitive.

- As mentioned before, on three occasions we discovered that one of the database replicas was different from the others in that Chubby cell. We found this problem because our system periodically takes checksums of all replicas and then compares them.
- Our upgrade script which is responsible for migrating cells from the 3DB version of Chubby to the Paxos version has failed several times for a variety of reasons. For example, it once failed because a basic Google program was not installed on one of our cells.
- We have encountered failures due to bugs in the underlying operating system. For example in our version of the Linux 2.4 kernel, when we try to flush a small file to disk, the call can hang for a long time if there are a lot of buffered writes to other files. This happens immediately after we write a database snapshot to disk. In this case, we observed that it could take several seconds for the kernel to flush an unrelated small write to the Paxos log. Our workaround is to write *all* large files in small chunks, with a flush to disk after each small chunk. While this hurts the performance of the write slightly, it protects the more critical log writes from unexpected delays.

A small number of failures in 100 machine years would be considered excellent behavior for most production systems. However, we consider the current failure rate too high for Chubby and we have determined that we need to reduce it further.

Three of the failures occurred during upgrade (or rollback). Every time we encountered a problem during upgrade, we updated our upgrade script accordingly. Once a cell is upgraded, this type of failure will disappear.

Two of the failures were from bugs that have since been fixed. To reduce the probability of other bugs, we continue to improve and run the Chubby verification test outlined earlier.

Two of our unexpected problems relate to operator error during rollout of a new release and caused us to lose data. At Google, the day-to-day monitoring and management of our systems is done by system operators. While they are very competent, they are usually not part of the development team that built the system, and therefore not familiar with its intricate details. This may lead to the occasional operator error in unforseen situations. We now rely on carefully written and well-tested scripts to automate rollout and minimize operator involvement. As a result our most recent major release of Chubby was rolled out across hundreds of machines without incident, while serving life traffic.

One of the failures was due to memory corruption. Because our system is log-structured and maintains several days of log data and snapshots, it was possible to replay the database upto the exact point at which the problem appears. We were able to verify that our logs were correct and conclude that the memory corruption occurred from errant software or due to hardware problems. We added additional checksum data to detect this type of problem in the future and will crash a replica when it detects this problem.

8 Measurements

The initial goal of our system was to replace 3DB with our own database. Thus our system had to demonstrate equal or superior performance relative to 3DB. We measured the performance of a complete Chubby system (clients, server, including network latency) using our fault-tolerant replicated database. We also benchmarked this system against an identical system based on 3DB (see Table 1). For our tests, we ran two copies of Chubby on the same set of 5 servers (typical Pentium®-class machines). One copy of Chubby used our database while the other copy used 3DB. We ran Chubby clients on workstations to generate load on the servers. For our tests, we measured total system throughput. Each call includes the Chubby client, the network, the Chubby server and our fault-tolerant database. While this test underestimates the performance of our database, it gives a sense of the full system throughput of a system based on Paxos.

Test	# workers	file size (bytes)	Paxos-Chubby (100MB DB)	3DB-Chubby (small database)	Comparison
Ops/s Throughput	1	5	91 ops/sec	75 ops/sec	1.2x
Ops/s Throughput	10	5	490 ops/sec	134 ops/sec	3.7x
Ops/s Throughput	20	5	640 ops/sec	178 ops/sec	3.6x
MB/s Throughput	1	8 KB	345 KB/s	172 KB/s	2x
MB/s Throughput	4	8 KB	777 - 949 KB/s	217 KB/s	3.6 - 4.4x
MB/s Throughput	1	32 KB	672 - 822 KB/s	338 KB/s	2.0 - 2.4x

Table 1: Comparing our system with 3DB (higher numbers are better).

Even though read requests to Chubby dominate in practice, we designed our tests to be write intensive. This is because read requests are completely handled on the master, which typically has a lease, and do not exercise the Paxos algorithm.

In our test, each worker repeatedly creates a file in Chubby and waits for Chubby to return before creating the file again. Thus each operation makes one write call to the underlying database. If the contents of the file are small and there is a single worker, the test measures the latency of the system. If the contents of the file are large, the test measures the throughput of the system in MB/s. By using multiple concurrent workers, we were also able to measure the throughput of the system in submissions/s.

All tests with more than one worker show the effect of batching a collection of submitted values. It should be possible to achieve some speedup with 3DB by bundling a collection of updates in a database transaction. The last two throughput tests show the effect of taking snapshots. This system was configured to take a snapshot whenever the replicated log size exceeded 100 MB. In these two tests, the system takes snapshots roughly every 100 seconds. When taking a snapshot, the system makes another copy of the database and writes it to disk. As a result, its performance temporarily drops off.

Our system is by no means optimized for performance, and we believe that there is a lot of room to make it faster. However, given the performance improvement over 3DB, further optimizations are not a priority at this time.

9 Summary and open problems

We have described our implementation of a fault-tolerant database, based on the Paxos consensus algorithm. Despite the large body of literature in the field, algorithms dating back more then 15 years, and experience of our team (one of us has designed a similar system before and the others have built other types of complex systems in the past), it was significantly harder to build this system then originally anticipated. We attribute this to several shortcomings in the field:

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.
- The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms.
- The fault-tolerance computing community has not paid enough attention to testing, a key ingredient for building fault-tolerant systems.

As a result, the core algorithms work remains relatively theoretical and is not as accessible to a larger computing community as it could be. We believe that in order to make a greater impact, researchers in the field should focus on addressing these shortcomings.

In contrast, consider the field of compiler construction. Though concepts in that field are complex, they have been made accessible to a wide audience. Industrial-strength parsing tools such `yacc` [6] appeared not too long after the theory of parsing was well-understood. Not only are there now many front-end tools such as `ANTLR` [15] or `CoCo/R` [13]; but there are also tree-rewriting tools helping with optimizations and instruction selection, assemblers helping with binary code generation, and so forth. Thus, in this area of software engineering, an entire family of tools has emerged, making the construction of a compiler significantly easier or at least less error-prone. Disciplines within the field of compiler construction, such as parsing, which were once at the cutting edge of research, are now considered "solved" and are routinely taught at the undergraduate level in many schools.

It appears that the fault-tolerant distributed computing community has not developed the tools and know-how to close the gaps between theory and practice with the same vigor as for instance the compiler community. Our experience suggests that these gaps are non-trivial and that they merit attention by the research community.

10 Acknowledgments

Many people at Google helped us with this project. Mike Burrows who implemented Chubby suggested that we replace 3DB with a Paxos-based system. He and Sharon Perl reviewed our designs and provided excellent feedback. They introduced us to the mechanism for handling disk corruptions and suggested that we implement master leases. Michal Cierniak ported the original state machine compiler from Perl to C++ and made substantial modifications (it is now being used elsewhere at Google as well). Vadim Furman helped us write the Chubby verification test. Salim Virji and his team were responsible for the roll-out of our system across Google data centers.

Mike Burrows, Bill Coughran, Gregory Eitzman, Peter Mckenzie, Sharon Perl, Rob Pike, David Presotto, Sean Quinlan, and Salim Virji reviewed earlier versions of this paper and provided valuable feedback.

References

[1] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 335-350

[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 205-218

[3] Cristian, F. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing 4*, 4 (1991), 175–188.

[4] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Dec. 2003), pp. 29–43.

[5] Gray, C., Cheriton, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (1989), pp. 202–210.

[6] Johnson, S. C. Yacc: Yet another compiler-compiler.

[7] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press.* 1995.

[8] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[9] Lamport, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[10] Lampson, B. W. How to build a highly available system using consensus. In *10th International Workshop on Distributed Algorithms (WDAG 96)* (1996), Babaoglu and Marzullo, Eds., vol. 1151, Springer-Verlag, Berlin Germany, pp. 1–17.

[11] Lee, E. K., and Thekkath, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.

[12] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004), pp. 105–120.

[13] Moessenboeck, H. A generator for production quality compilers. In *Proceedings of the 3rd International Workshop on Compiler Compilers - Lecture Notes in Computer Science 477* (Berlin, Heidelberg, New York, Tokyo, 1990), Springer-Verlag, pp. 42–55.

[14] Oki, Brian M., and Liskov, Barbara H. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[15] Parr, T. J., and QUONG, R. W. Antlr: A predicated-ll(k) parser generator. *Software–Practice and Experience 25*, 7 (JULY 1995), 789–810.

[16] Prisco, R. D., Lampson, B. W., and Lynch, N. A. Revisiting the paxos algorithm. In *11th International Workshop on Distributed Algorithms (WDAG 96)* (1997), pp. 111–125.

[17] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (1990), 299–319.

[18] von Neumann, J. Probabilistic logics and synthesis of reliable organisms from unreliable components. *Automata Studies* (1956), 43–98.

Chapter 9

Bitcoin: A Peer-to-Peer Electronic Cash System

Authors:
Satoshi Nakamoto

Publication year: 2008

Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

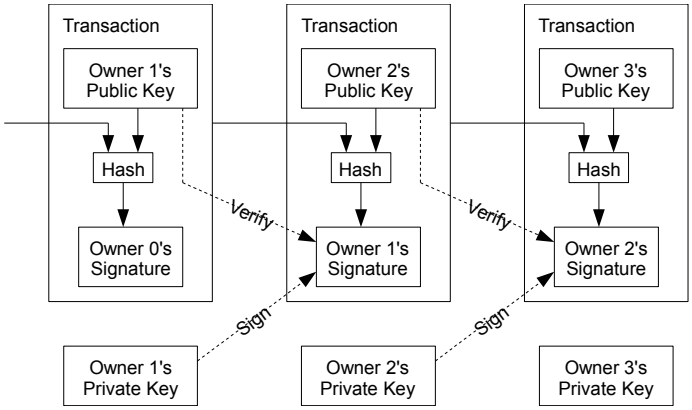
1. Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. Transactions that are computationally impractical to reverse would protect sellers from fraud, and routine escrow mechanisms could easily be implemented to protect buyers. In this paper, we propose a solution to the double-spending problem using a peer-to-peer distributed timestamp server to generate computational proof of the chronological order of transactions. The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.

2. Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.

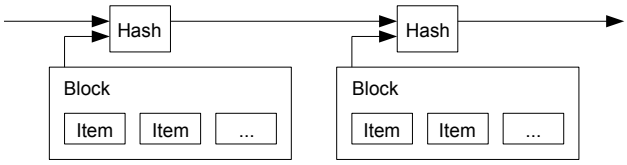


The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint, that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double-spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank.

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced [1], and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received.

3. Timestamp Server

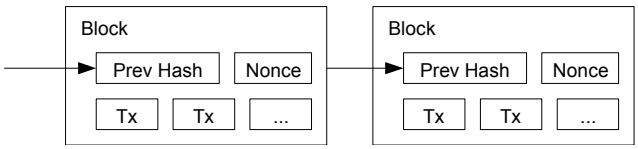
The solution we propose begins with a timestamp server. A timestamp server works by taking a hash of a block of items to be timestamped and widely publishing the hash, such as in a newspaper or Usenet post [2-5]. The timestamp proves that the data must have existed at the time, obviously, in order to get into the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it.



4. Proof-of-Work

To implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof-of-work system similar to Adam Back's Hashcash [6], rather than newspaper or Usenet posts. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash.

For our timestamp network, we implement the proof-of-work by incrementing a nonce in the block until a value is found that gives the block's hash the required zero bits. Once the CPU effort has been expended to make it satisfy the proof-of-work, the block cannot be changed without redoing the work. As later blocks are chained after it, the work to change the block would include redoing all the blocks after it.



The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes. We will show later that the probability of a slower attacker catching up diminishes exponentially as subsequent blocks are added.

To compensate for increasing hardware speed and varying interest in running nodes over time, the proof-of-work difficulty is determined by a moving average targeting an average number of blocks per hour. If they're generated too fast, the difficulty increases.

5. Network

The steps to run the network are as follows:

1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Nodes always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next proof-of-work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one.

New transaction broadcasts do not necessarily need to reach all nodes. As long as they reach many nodes, they will get into a block before long. Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

6. Incentive

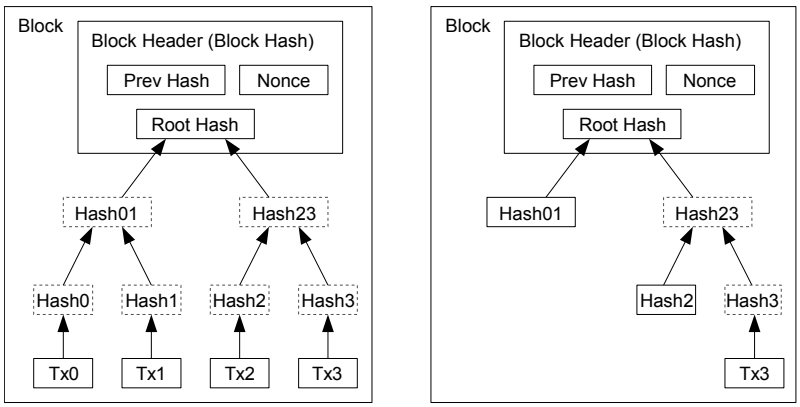
By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.

The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free.

The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable to play by the rules, such rules that favour him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth.

7. Reclaiming Disk Space

Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree [7][2][5], with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored.



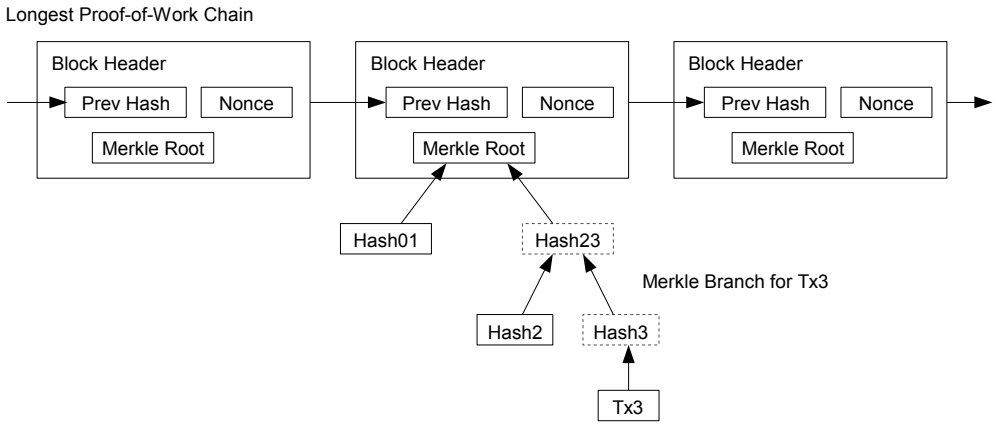
Transactions Hashed in a Merkle Tree

After Pruning Tx0-2 from the Block

A block header with no transactions would be about 80 bytes. If we suppose blocks are generated every 10 minutes, 80 bytes * 6 * 24 * 365 = 4.2MB per year. With computer systems typically selling with 2GB of RAM as of 2008, and Moore's Law predicting current growth of 1.2GB per year, storage should not be a problem even if the block headers must be kept in memory.

8. Simplified Payment Verification

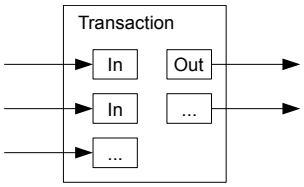
It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.



As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network. One strategy to protect against this would be to accept alerts from network nodes when they detect an invalid block, prompting the user's software to download the full block and alerted transactions to confirm the inconsistency. Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.

9. Combining and Splitting Value

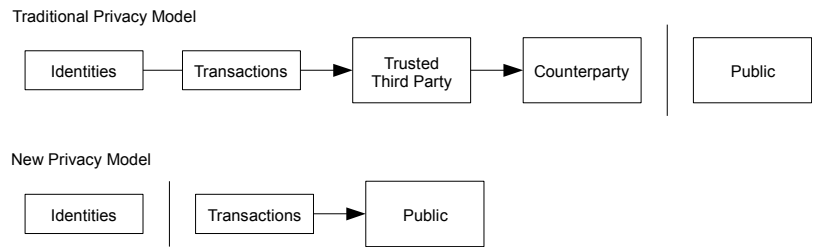
Although it would be possible to handle coins individually, it would be unwieldy to make a separate transaction for every cent in a transfer. To allow value to be split and combined, transactions contain multiple inputs and outputs. Normally there will be either a single input from a larger previous transaction or multiple inputs combining smaller amounts, and at most two outputs: one for the payment, and one returning the change, if any, back to the sender.



It should be noted that fan-out, where a transaction depends on several transactions, and those transactions depend on many more, is not a problem here. There is never the need to extract a complete standalone copy of a transaction's history.

10. Privacy

The traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party. The necessity to announce all transactions publicly precludes this method, but privacy can still be maintained by breaking the flow of information in another place: by keeping public keys anonymous. The public can see that someone is sending an amount to someone else, but without information linking the transaction to anyone. This is similar to the level of information released by stock exchanges, where the time and size of individual trades, the "tape", is made public, but without telling who the parties were.



As an additional firewall, a new key pair should be used for each transaction to keep them from being linked to a common owner. Some linking is still unavoidable with multi-input transactions, which necessarily reveal that their inputs were owned by the same owner. The risk is that if the owner of a key is revealed, linking could reveal other transactions that belonged to the same owner.

11. Calculations

We consider the scenario of an attacker trying to generate an alternate chain faster than the honest chain. Even if this is accomplished, it does not throw the system open to arbitrary changes, such as creating value out of thin air or taking money that never belonged to the attacker. Nodes are not going to accept an invalid transaction as payment, and honest nodes will never accept a block containing them. An attacker can only try to change one of his own transactions to take back money he recently spent.

The race between the honest chain and an attacker chain can be characterized as a Binomial Random Walk. The success event is the honest chain being extended by one block, increasing its lead by +1, and the failure event is the attacker's chain being extended by one block, reducing the gap by -1.

The probability of an attacker catching up from a given deficit is analogous to a Gambler's Ruin problem. Suppose a gambler with unlimited credit starts at a deficit and plays potentially an infinite number of trials to try to reach breakeven. We can calculate the probability he ever reaches breakeven, or that an attacker ever catches up with the honest chain, as follows [8]:

p = probability an honest node finds the next block
q = probability the attacker finds the next block
q_z = probability the attacker will ever catch up from z blocks behind

$$q_z = \begin{cases} 1 & \text{if } p \le q \\ (q/p)^z & \text{if } p > q \end{cases}$$

Given our assumption that $p > q$, the probability drops exponentially as the number of blocks the attacker has to catch up with increases. With the odds against him, if he doesn't make a lucky lunge forward early on, his chances become vanishingly small as he falls further behind.

We now consider how long the recipient of a new transaction needs to wait before being sufficiently certain the sender can't change the transaction. We assume the sender is an attacker who wants to make the recipient believe he paid him for a while, then switch it to pay back to himself after some time has passed. The receiver will be alerted when that happens, but the sender hopes it will be too late.

The receiver generates a new key pair and gives the public key to the sender shortly before signing. This prevents the sender from preparing a chain of blocks ahead of time by working on it continuously until he is lucky enough to get far enough ahead, then executing the transaction at that moment. Once the transaction is sent, the dishonest sender starts working in secret on a parallel chain containing an alternate version of his transaction.

The recipient waits until the transaction has been added to a block and z blocks have been linked after it. He doesn't know the exact amount of progress the attacker has made, but assuming the honest blocks took the average expected time per block, the attacker's potential progress will be a Poisson distribution with expected value:

$$\lambda = z \frac{q}{p}$$

To get the probability the attacker could still catch up now, we multiply the Poisson density for each amount of progress he could have made by the probability he could catch up from that point:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{Bmatrix} (q/p)^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{Bmatrix}$$

Rearranging to avoid summing the infinite tail of the distribution...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

Converting to C code...

```
#include <math.h>
double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

Running some results, we can see the probability drop off exponentially with z.

```
q=0.1
z=0    P=1.0000000
z=1    P=0.2045873
z=2    P=0.0509779
z=3    P=0.0131722
z=4    P=0.0034552
z=5    P=0.0009137
z=6    P=0.0002428
z=7    P=0.0000647
z=8    P=0.0000173
z=9    P=0.0000046
z=10   P=0.0000012

q=0.3
z=0    P=1.0000000
z=5    P=0.1773523
z=10   P=0.0416605
z=15   P=0.0101008
z=20   P=0.0024804
z=25   P=0.0006132
z=30   P=0.0001522
z=35   P=0.0000379
z=40   P=0.0000095
z=45   P=0.0000024
z=50   P=0.0000006
```

Solving for P less than 0.1%...

```
P < 0.001
q=0.10   z=5
q=0.15   z=8
q=0.20   z=11
q=0.25   z=15
q=0.30   z=24
q=0.35   z=41
q=0.40   z=89
q=0.45   z=340
```

12. Conclusion

We have proposed a system for electronic transactions without relying on trust. We started with the usual framework of coins made from digital signatures, which provides strong control of ownership, but is incomplete without a way to prevent double-spending. To solve this, we proposed a peer-to-peer network using proof-of-work to record a public history of transactions that quickly becomes computationally impractical for an attacker to change if honest nodes control a majority of CPU power. The network is robust in its unstructured simplicity. Nodes work all at once with little coordination. They do not need to be identified, since messages are not routed to any particular place and only need to be delivered on a best effort basis. Nodes can leave and rejoin the network at will, accepting the proof-of-work chain as proof of what happened while they were gone. They vote with their CPU power, expressing their acceptance of valid blocks by working on extending them and rejecting invalid blocks by refusing to work on them. Any needed rules and incentives can be enforced with this consensus mechanism.

References

[1] W. Dai, "b-money," http://www.weidai.com/bmoney.txt, 1998.

[2] H. Massias, X.S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirements," In *20th Symposium on Information Theory in the Benelux*, May 1999.

[3] S. Haber, W.S. Stornetta, "How to time-stamp a digital document," In *Journal of Cryptology*, vol 3, no 2, pages 99-111, 1991.

[4] D. Bayer, S. Haber, W.S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," In *Sequences II: Methods in Communication, Security and Computer Science*, pages 329-334, 1993.

[5] S. Haber, W.S. Stornetta, "Secure names for bit-strings," In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 28-35, April 1997.

[6] A. Back, "Hashcash - a denial of service counter-measure," http://www.hashcash.org/papers/hashcash.pdf, 2002.

[7] R.C. Merkle, "Protocols for public key cryptosystems," In *Proc. 1980 Symposium on Security and Privacy*, IEEE Computer Society, pages 122-133, April 1980.

[8] W. Feller, "An introduction to probability theory and its applications," 1957.